CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS REÇUE

Computation of Weight Multiplicities in Representations of Lie Algebras

Murray Ronald Bremner

A Thesis

in

the Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

March 1984

ISBN 0-315-30676-9

Abstract

Computation of Weight Multiplicities
in Representations of Lie Algebras

Murray Ronald Bremner

This thesis presents a Pascal implementation of a fast recursion formula for the multiplicities of dominant weights in finite-dimensional representations of simple Lie algebras over the complex numbers. The fast recursion formula (a modified version of the Freudenthal formula) and a new algorithm for the direct computation of the dominant weights were developed by R. V. Moody and J. Patera (Fast recursion formula for weight multiplicities, Bull. Amer. Math. Soc., 7, 1982, 237-242).

Two programs are included with this thesis. The first, interactive, implements the dominant weight algorithm and the fast recursion formula; the user specifies a Lie algebra and a representation (i.e. a highest weight) and the program computes the dominant weights and their multiplicities. The second, non-interactive, incorporates many of the procedures and functions of the first; the user specifies a Lie algebra and a congruence class of representations, as well as the number of representations to be processed, and the program then generates the desired number of highest weights (by increasing level) and computes

an upper-triangular table containing the dominant weights and their multiplicities for each representation. All computations are exact.

This thesis is an application of computer science to a computational problem of pure mathematics and theoretical physics.

Acknowledgements

I would like to thank Prof. J. McKay, Prof. R. V. Moody, and Dr. J. Patera for their advice and encouragement throughout my work on this thesis, and in addition P. Dubois, R. Funk, C. Grossner, L. Lam, and L. Thiel for their assistance with various matters.

I would also like to express my appreciation to my family.

This research was supported in part by Concordia University and the Government of Quebec.

Table of contents

Notation

L	a Lie algebra
F	a field
LxL	the Cartesian product of L with itself
[xy]	the commutator (i.e. the Lie bracket) of x and y
K	a subspace or subalgebra of L
V	a finite-dimensional vector space over F
End V	the ring of linear transformations from V to V
:=	is defined to be
gl(V)	the general linear algebra of V
Der A	the set of derivations of the F-algebra A
ad	the adjoint representation of L
I	an ideal of L
Z(L)	the center of L
[LL]	the derived algebra of L
r	a representation of L
$L^{(i)}$	the derived series of L
L^i	the descending central series of L
rad L	the radical of L
k(x,y)	the Killing form
rad k	the radical of k
N(K)	the normalizer of K
C(S)	the centralizer of the subset S of L
H	a Cartan subalgebra of L
L(a)	the root space of the root a
H^*	the dual space of H

R	the root system of L
(a,b)	the bilinear form on H^* corresponding to k
E	the real vector space spanned by R
$s(a)$	the reflection determined by the root a
$P(a)$	the fixed hyperplane of $s(a)$
$\langle b,a \rangle$	$:= 2(b,a)/(a,a)$
W	the Weyl group of R
$GL(E)$	the general linear group of E
n	the rank of the Lie algebra L
$D := \{a_1, \ldots, a_n\}$	a base of the root system R
$R+$	the set of positive roots relative to D
G	the weight lattice
$G+$	the set of dominant weights in G
$\{w_1, \ldots, w_n\}$	the fundamental weights
f.d.i.	finite-dimensional irreducible
$V(g)$	the weight space of the weight g
$m(g)$	the multiplicity of the weight g
$stab(g)$	the subgroup of W which stabilizes the weight g
$stab'(g)$	the subgroup of $GL(E)$ generated by $stab(g)$ and the negative of the identity transformation
O_j	the orbits of F under $stab'(g)$
x_j	the unique subdominant root in O
$[O_j]$	the size of the orbit O
δ_{ij}	the Kronecker delta: 1 if $i=j$, 0 otherwise

Chapter 1.

Introduction; Background in Lie theory

The Freudenthal formula (Freudenthal 1954) gives a constructive method for determining the multiplicities of weights of finite-dimensional representations of semisimple Lie algebras over the complex numbers. This problem, apart from its significance in pure mathematics, is important in the applications of Lie groups and algebras to theoretical physics (Loebl 1968, Weyl 1950, Wigner 1959). Several computer implementations of this formula have been discussed in the literature; the most well-known are those of Agrawala and Belinfante (1969), Krusemeyer (1971), Beck and Kolman (1972) (see also Kolman and Beck 1973a and 1973b), and McKay, Patera and Sankoff (1977). The Pascal implementation presented in this thesis is based on a modified version of the Freudenthal formula described and proved in the paper of Moody and Patera (1982). Two important new results are presented in that paper: first, an algorithm for the direct computation of the dominant weights of a representation, and second, a more efficient multiplicity formula derived from Freudenthal's formula by exploiting the Weyl group to 'compress' it as much as possible. This thesis presents the first complete implementation of both the dominant weight algorithm and the modified Freudenthal formula. This work is intended as a contribution to the field of exact

computation as applied to pure mathematics and theoretical physics.

The remainder of this chapter consists of an outline of the Lie-theoretic background of this project; the account is based on that of Humphreys (1972). The reader is assumed to be familiar with basic algebra at the undergraduate level as covered in e.g. Herstein (1975) or Jacobson (1974).

A vector space L over a field F, with a binary operation LxL --> L (denoted (x,y) --> [xy]), is a 'Lie algebra' over F if

1) the bracket operation [xy] is bilinear,

2) for all x in L, [xx] = 0, and

3) for all x, y, z in L, we have the 'Jacobi identity': [x[yz]] + [y[zx]] + [z[xy]] = 0.

Note that properties 1) and 2) together imply that [xy] = -[yx] for all x, y in L.

A Lie algebra L satisfying [xy] = 0 for all x, y in L is an 'abelian' Lie algebra. A subspace K of a Lie algebra L is a (Lie) 'subalgebra' of L if [xy] is in K for all x, y in K.

Let V be a finite-dimensional vector space over F, and let End V be the ring of linear transformations from V to V. If we define [xy] := xy - yx for all x, y in End V then we can view End V as a Lie algebra, the 'general linear algebra' of V, denoted gl(V). Any subalgebra of a general linear algebra is a 'linear Lie algebra'.

Given a field F, by 'F-algebra' (not necessarily

associative) we mean a vector space A over F with a bilinear operation AxA --> A (denoted (a,b) --> a.b). A linear map d: A --> A satisfying d(a.b) = a.d(b) + d(a).b (the 'derivative rule') is a 'derivation' of A. Since the Lie bracket of two derivations is a derivation, the subspace Der A of End A consisting of all derivations of A is a Lie subalgebra of gl(A).

Given a Lie algebra L and any x in L we denote by 'ad x' the endomorphism of L which sends y --> [xy] for all y in L; ad x is a derivation of L. Derivations of L of the form ad x for some x in L are 'inner'; all others are 'outer'. The map L --> Der L sending x --> ad x is the 'adjoint representation' of L.

A subspace I of a Lie algebra L is an 'ideal' of L if [xy] is in I whenever x is in L and y is in I. Ideals arise as kernels of homomorphisms of Lie algebras. Two important examples of ideals are the 'center' of L,

$$Z(L) := \{z \text{ in } L: [xz] = 0 \text{ for all } x \text{ in } L\},$$

and the 'derived algebra' of L,

$$[LL] := \langle [xy]: x, y \text{ in } L \rangle,$$

i.e. the subspace of L spanned by all [xy] with x, y in L. The following are equivalent: L is abelian; Z(L) = L; [LL] = {0}. If L has no ideals except itself and {0}, and if [LL] ≠ 0 (i.e. L is not abelian), then L is 'simple'. If L is simple then Z(L) = {0} and [LL] = L. If I is an ideal of L, we construct the 'quotient algebra' L/I by defining L/I to be the vector space quotient L/I with the

Lie bracket $[x+I,y+I] := [xy] + I$.

Let L and L' be Lie algebras over F. A linear transformation $h: L \to L'$ is a 'homomorphism' if $h([xy]) = [h(x)h(y)]$ for all x, y in L. Such an h is a 'monomorphism' (injective, one-to-one) if $\ker h = \{0\}$, an 'epimorphism' (surjective, onto) if $\text{im } h = L'$, and an 'isomorphism' if both these conditions hold. Ker h is an ideal of L, and im h is a subalgebra of L'. There is a natural one-to-one correspondence between homomorphisms and ideals; to the homomorphism h corresponds the ideal ker h and to the ideal I corresponds the canonical (surjective) projection $L \to L/I$ given by $x \to x+I$.

All of the standard homomorphism theorems follow through for Lie algebras. 1) If $h: L \to L'$ is a homomorphism of Lie algebras, then $L / \ker h$ is isomorphic to im h; and if I is any ideal contained in ker h, then there exists a unique homomorphism $h': L/I \to L'$ such that $h'p = h$ where p is the canonical projection of L onto L/I. 2) If $I \subseteq J$ are ideals of L, then J/I is an ideal of L/I and $(L/I)/(J/I)$ is isomorphic to L/J. 3) If I and J are ideals of L, then $(I+J)/J$ is isomorphic to $I/(I \cap J)$.

A 'representation' of a Lie algebra L is a homomorphism $r: L \to gl(V)$ for some vector space V over F. A representation is 'faithful' if it is a monomorphism. It can be shown that every finite-dimensional Lie algebra has a faithful finite-dimensional representation; this is the Ado-Iwasawa theorem (see Jacobson 1962).

We define a sequence of ideals of L (the 'derived series') by $L^{(0)} := L$ and $L^{(i)} := [L^{(i-1)} L^{(i-1)}]$ for $i \geq 1$. If $L^{(i)} = \{0\}$ for some i then we call L 'solvable'. Abelian Lie algebras are solvable; simple Lie algebras are not solvable. If L is solvable, then so are all subalgebras and homomorphic images of L. There is a unique maximal solvable ideal in any Lie algebra L, the 'radical' of L, denoted rad L. If rad $L = \{0\}$, then L is 'semisimple'; equivalently L has no abelian ideals other than $\{0\}$. If the field F has characteristic 0, then L is semisimple if and only if it is the direct sum of simple ideals. If L is semi-simple, then $L = [LL]$ and all ideals and homomorphic images of L are also semisimple.

We define another sequence of ideals of L (the 'descending central series') by $L^0 := L$ and $L^i := [LL^{i-1}]$ for $i \geq 1$. L is 'nilpotent' if $L^i = \{0\}$ for some i. Abelian Lie algebras are nilpotent. If L is nilpotent then so are all subalgebras and homomorphic images of L. Since $L^{(i)} \subseteq L^i$ for all i, nilpotent Lie algebras are solvable; the converse however is false.

In the following we assume that L is a semisimple Lie algebra over the complex numbers.

For x, y in L we define k(x,y) to be the trace of ad x.ad y; k is a symmetric bilinear form on ., the 'Killing form'. L is semisimple if and only if the Killing form is non-degenerate, i.e. if the 'radical' of k,

rad k := {x in L: k(x,y) = 0 for all y in L}

is {0}.

A vector space V with an operation LxV --> V (denoted (x,v) --> x.v) is an 'L-module' if

1) (ax + by).v = a(x.v) + b(y.v),

2) x.(av + bw) = a(x.v) + b(x.w), and

3) [xy].v = xy.v - yx.v,

all three conditions holding for all x, y in L, v, w in V, and a, b in F. If r: L --> gl(V) is a representation of L, then V may be viewed as an L-module by the action x.v := r(x)v; conversely, given an L-module V, the equation r(x)v := x.v defines a representation r: L --> gl(V).

A 'homomorphism' of the L-modules V and W is a linear map h: V --> W such that h(x.v) = x.h(v) for all x in L, v in V; ker h is a submodule of V. If h is an isomorphism of vector spaces, then we call h an isomorphism of the L-modules V and W; V and W 'afford equivalent representations' of L. An L-module is 'irreducible' if it has exactly two distinct L-submodules (itself and {0}); in particular the zero-dimensional vector space is not regarded as an irreducible L-module. The L-module V is 'completely reducible' if V is the direct sum of irreducible L-modules. A fundamental theorem due to H. Weyl states that if r: L --> gl(V) is a finite-dimensional representation of a semisimple Lie algebra L, then r is completely reducible (see Humphreys 1972).

The 'normalizer' of a subspace (in particular a subalgebra) K of L is

$N(K) := \{x \text{ in } L: [xK] \subseteq K\}$;

by the Jacobi identity, $N(K)$ is a subalgebra of L. If K is a subalgebra of L, then $N(K)$ is the largest subalgebra of L which contains L as an ideal. If $K = N(K)$ then K is 'self-normalizing'. The 'centralizer' of a subset S of L is

$C(S) := \{x \text{ in } L: [xS] = \{0\}\}$;

$C(S)$ is also a subalgebra of L.

A subalgebra H of L is a 'Cartan subalgebra' if H is nilpotent and self-normalizing. Every semisimple Lie algebra has at least one Cartan subalgebra; the Cartan subalgebras of semisimple Lie algebras are abelian.

Let H be a Cartan subalgebra of L. It can be shown that L is the direct sum of the subspaces

$L(a) := \{x \text{ in } L: [hx] = a(h)x \text{ for all } h \text{ in } H\}$,

where a ranges over H^*, the dual space of H. Note that $L(0) = C(H)$ (the centralizer of H), which contains H since Cartan subalgebras are abelian; in fact $L(0) = H$ since it can be shown that $C(H) = H$ for any Cartan subalgebra H. We denote by R the set of all non-zero a in H^* for which $L(a) \neq \{0\}$; the elements of R are the 'roots' of L relative to the Cartan subalgebra H. We then have the 'root space decomposition' of L as a direct sum of vector spaces

$$L = H \oplus \bigoplus_{a \in R} L(a).$$

It can be shown that the restriction of the Killing form to H is non-degenerate. This allows us to identify H with H^* as follows: to a in H^* corresponds the unique element $t(a)$ in H satisfying $a(h) = k(t(a),h)$ for all h in

H. In particular R corresponds to the subset {t(a): a in R} of H. We can transfer the Killing form to H* by defining (a,b) := k(t(a),t(b)) for all a, b in H*. We denote by E the real vector space spanned by the roots R over the real numbers. The Killing form extends canonically to E and is positive definite.

We have the following results:

1) R is finite, spans E, and does not contain 0;

2) if a is in R then -a is in R but no other scalar multiple of a is a root;

3) if a, b are in R then b - (2(b,a)/(a,a))a is in R; and

4) if a, b are in R then 2(b,a)/(a,a) is an integer.

Given a finite-dimensional real vector space (with a positive definite symmetric bilinear form), a set of vectors satisfying these four axioms is called a 'root system'.

A 'reflection' in E is an invertible linear transformation leaving pointwise fixed some hyperplane (subspace of codimension 1) and sending any vector orthogonal to that hyperplane into its negative. Reflections preserve the scalar product on E. Any non-zero vector a in E (in particular a root) determines a reflection s(a) with reflecting hyperplane P(a) := {b in E: (b,a) = 0}. Now s(a): b --> b - (2(b,a)/(a,a))a since it sends a to -a and fixes all points in P(a). We abbreviate 2(b,a)/(a,a) to <b,a>; note that this is linear only in the first variable.

We denote by W the subgroup of GL(E) (the general

linear group of E) generated by the reflections s(a) for a in R. By axiom 3) for a root system, W permutes R, which by axiom 1) is finite. Hence we can identify W with a subgroup of the symmetric group on the root system R; in particular W is finite, and is known as the 'Weyl group' of R. We call $n := \dim E$ the 'rank' of the root system.

A subset D of R is a 'base' of R if

1) D is a basis of E, and

2) each root a in R can be written as $a = \sum_{b \in D} c_b b$ with integral coefficients c_b either all non-negative or all non-positive.

It can be shown that every root system R has a base D; the roots in D are 'simple'. Since D is a basis of E the expression given in 2) above is unique. Hence we can define the 'height' of a root a (relative to D) by $ht(a) := \sum_{b \in D} c_b$. If all c_b are non-negative (resp. non-positive) a is 'positive' (resp. 'negative'). We denote by R^+ the set of all positive roots with respect to the base D.

A root system R is 'irreducible' if it cannot be partitioned into the union of two proper subsets such that each root in one subset is orthogonal to each root in the other. It can be shown that R is irreducible if and only if D cannot be partitioned in this way. If R is an irreducible root system, then at most two root lengths occur in R, and all roots of a given length are conjugate under the Weyl group W. (Vectors l and l' are said to be 'conjugate' under W if $l' = wl$ for some w in W). If there are two distinct

root lengths, we speak of 'long' and 'short' roots. (See Humphreys 1972, pp. 52-53.)

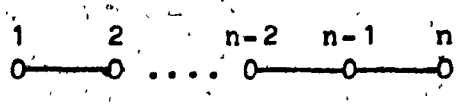
Given a fixed numbering of the n simple roots $a_1, \ldots, a_n$, the matrix $(\langle a_i, a_j \rangle)$ is the 'Cartan matrix' of R; its entries are the 'Cartan integers'. The Cartan matrix of R determines R up to isomorphism. Throughout this thesis we assume the numbering of the simple roots used by Bourbaki (1968); this convention is also used by Humphreys (1972).

Let R be an arbitrary root system. If a, b in R are distinct positive roots, then it can be shown that $\langle a,b\rangle\langle b,a\rangle = 0, 1, 2,$ or 3; the product is 0 if and only if a and b are orthogonal and 1 if and only if a and b have the same length. The 'Coxeter graph' of R is a graph having $n = \dim E$ vertices, the i-th joined to the j-th ($i \neq j$) by $\langle a_i, a_j \rangle\langle a_j, a_i \rangle$ edges. Wherever a double or triple edge occurs in the Coxeter graph of R, we can add an arrow pointing to the shorter of the two roots; the arrow can be regarded as a 'less than' sign referring to the root lengths. The resulting diagram is the 'Dynkin diagram' of R.

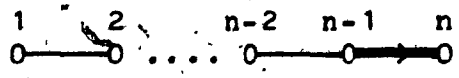
A root system R is irreducible if and only if its Coxeter graph (Dynkin diagram) is connected. Any root system R decomposes uniquely as the union of irreducible root systems R_i in subspaces E_i of E such that $E = E_1 \oplus \ldots \oplus E_m$ (orthogonal direct sum). Hence it is sufficient to classify irreducible root systems, or equivalently the connected Dynkin diagrams.

It can be shown that if R is an irreducible root system of rank n then its Dynkin diagram is one of the following (n vertices in each case):

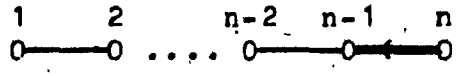
An ($n \geq 1$)



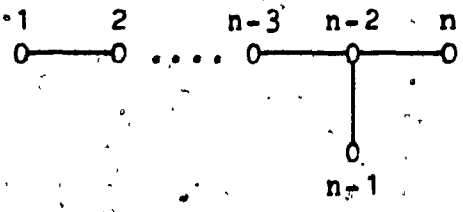
Bn ($n \geq 2$)



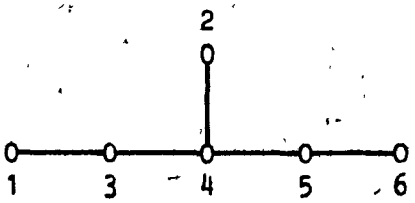
Cn ($n \geq 3$)



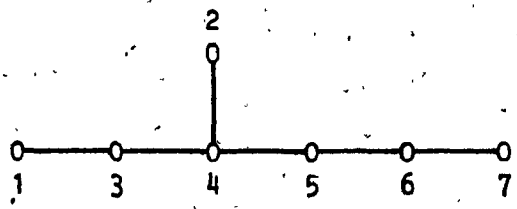
Dn ($n \geq 4$)

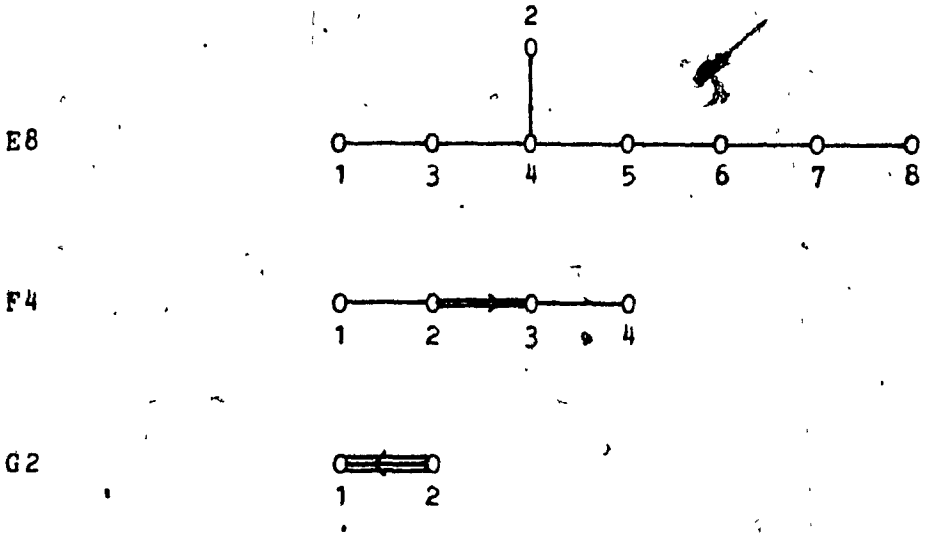


E6



E7





We put restrictions on the rank in cases B, C, and D in order to avoid repetitions: A1, B1, C1, and D1 are isomorphic; similarly B2 and C2; D2 and A1xA1; and D3 and A3.

Let R be a root system in a Euclidean space E, with Weyl group W. We denote by G the set of all g in E for which $\langle g,a \rangle$ is integral for all a in R; the elements of G are 'weights', and G is the 'weight lattice'. G is a subgroup of E which includes R. The subgroup of G-generated by R is the 'root lattice'. If we fix a base D of R, g is in G if and only if $\langle g,a \rangle$ is integral for all a in D; we call g 'dominant' if $\langle g,a \rangle \geq 0$ for all a in D and 'strongly dominant' if $\langle g,a \rangle > 0$ for all a in D. We denote by G+ the set of all dominant weights in G.

Let $w_1, \ldots, w_n$ be the dual basis (relative to the scalar product on E) defined by $\langle w_i, a_j \rangle = \delta_{ij}$. Since all the $\langle w_i, a_j \rangle$ are non-negative integers, the w_i are dominant

weights; they are the 'fundamental weights' (relative to D). The Cartan matrix expresses the change of basis: $a_i = \sum_{j=1}^{n} \langle a_i, a_j \rangle w_j$.

Each weight in G is conjugate under W to exactly one dominant weight. Given some g in G+, the number of dominant weights $g' \leq g$ is finite (where we define $g' \leq g$ if and only if $g - g'$ is a sum of positive roots or $g - g' = 0$).

Chapter 2.

Representation theory and the Freudenthal formula

Recall that a finite-dimensional representation r of a Lie algebra L is a homomorphism r: L --> gl(V) of L into the Lie algebra gl(V) of linear transformations on some finite-dimensional vector space V. The L-module V is 'irreducible' if it has exactly two submodules (V and {0}); in other words if the only r-invariant subspaces U of V (i.e. subspaces U satisfying $r(L)U \subseteq U$) are V and {0}. The L-module V is 'completely reducible' if V is a direct sum of irreducible L-submodules. We know by Weyl's theorem that every finite-dimensional representation of a semisimple Lie algebra is completely reducible.

We now restrict our attention to finite-dimensional irreducible (f.d.i.) representations (r, V) of a simple Lie algebra L over the complex numbers. We denote by H a fixed Cartan subalgebra of L, H^* its dual space, R the root system of L, $D = \{a_1, \ldots, a_n\}$ a base of R, and $\mathcal{W}$ the Weyl group. A linear function g in H^* is a 'weight' of V if there exists a nonzero vector v in V such that r(h)v = g(h)v for all h in H; i.e. v is simultaneously an eigenvector for all the linear transformations r(h) with corresponding eigenvectors g(h). Such a vector v is a 'weight vector' for the weight g; the set of all weight vectors for a given weight g together with the zero vector form an L-submodule of V, the

'weight space' of g, denoted V(g). It can be shown that V is the direct sum of its weight spaces; since V is finite-dimensional the number of weights is finite. It can also be shown that for any weight g of an f.d.i. L-module V and for any simple root a_i, $\langle g, a_i \rangle$ is integral; hence the weights associated with a particular representation are weights in the sense of the abstract theory discussed in the previous chapter.

As an example we may consider the adjoint representation of L, ad: L --> gl(L) where ad x: y --> [xy]. In this case the weights are the roots a in R together with the zero vector; each weight space V(a) (i.e. L(a) in the previous notation) has dimension 1 and the weight space V(0) (i.e. H) has dimension n.

The set of all weights for a representation r is the 'weight system' of r, and the dimension of the weight space V(g), denoted m(g), is the 'multiplicity' of the weight g.

Let V be any f.d.i. L-module. It can be shown that the weight system of V contains a unique 'highest weight', i.e. a weight h such that all other weights of V are of the form $g = h - \sum_{i=1}^{n} c_i a_i$, $c_i \geq 0$. (The sum $\sum_{i=1}^{n} c_i$ is the 'layer' of the weight.) In terms of the partial ordering defined at the end of the last chapter, we may write $g \leq h$ for all weights g of V. The highest weight of a representation always has multiplicity 1. Two f.d.i. L-modules are isomorphic if and only if they have the same highest weight. A basic result in representation theory states that the

highest weight h of an f.d.i. L-module V is a dominant weight, and conversely that given any dominant weight g there exists a f.d.i. L-module V with g as its highest weight. Thus there is a one-to-one correspondence between the set of dominant weights and the set of isomorphism classes of finite-dimensional irreducible L-modules.

The weight lattice G is partially ordered by the 'level' function: the level of a weight g is $2(g,d') + 1$ where d' is defined by the conditions $(d',a_i) = 1$ for all simple roots a_i. If we partition the complete weight system of a representation (with highest weight h) into levels using the level function, then the level of h is the number of levels.

Given a dominant weight h, the dimension of the corresponding L-module V(h) can be determined by the following product formula, the 'Weyl dimension formula' (see Humphreys 1972, p. 139):

$$\dim V(h) = \prod_{a \in R+} (h+d,a) \; / \; \prod_{a \in R+} (d,a),$$

where d is the sum of the fundamental weights (equivalently half the sum of the positive roots).

An important fact which will be needed in the next chapter is the following. Given an f.d.i. L-module V with highest weight h and any weight g of V, then the weights of V of the form $g + ra$ for some fixed root a and some integer r form a connected 'string' through g in the sense that there exist non-negative integers p and q such that $g + ra$ is a weight if and only if $-p \leq r \leq q$. Moreover, the

reflection $s(a)$ reverses this string, and $p - q = \langle g,a \rangle$.

Given an f.d.i. L-module V with highest weight h, we want to be able to determine which weights g occur in the weight system of the module, as well as the multiplicity of each weight. Now if g is any weight of V, and w is an arbitrary element of the Weyl group W, then wg is also a weight of V with the same multiplicity as g. Each weight of V is conjugate under W to a unique dominant weight; there is a straightforward method for determining the dominant weight conjugate to any given weight (see the 'dominant weight lemma' in the next chapter). Thus it suffices to be able to compute the dominant weights and their multiplicities. This fact is crucial, since the new weight algorithm in Moody and Patera (1982) computes only the dominant weights, and the programs presented in this thesis compute only the dominant weights and their multiplicities.

The most common method of determining the complete weight system, given the highest weight, is the algorithm of Dynkin (1957). The following explanation of this algorithm is based on the discussion in Belinfante and Kolman (1972). The weight system is divided into disjoint layers, layer 0 consisting of the highest weight alone. The layers are then generated inductively; to determine one layer, given all the previous layers, we consider the simple root strings passing through the weights in the previous layer. Let $g = g_1 w_1 + \ldots + g_n w_n$ be a weight in the previous layer, expressed relative to the basis of fundamental weights. We

know that for some non-negative integers p and q, the weight $g + ra_j$ is in the weight system if and only if $-p \leq r \leq q$, and that the 'top' and 'bottom' weights of the string have the property that $s(a_j)(g - pa_j) = g + qa_j$. Since $s(a_j)(g) = g - g_j a_j$ and $s(a_j)(a_j) = -a_j$ we then have $g - g_j a_j + pa_j = g + qa_j$ and so $p = q + g_j$. Since the weights lying above g in this string belong to previous layers, the value of q is known and the value of p is then easily computed. The weight $g - a_j$ will lie in the layer currently being determined if and only if p is positive.

One of the most widely-used formulas for computing weight multiplicities is that of Freudenthal (1954):

$$\{(h + d,h + d) - (g + d,g + d)\}m(g) = 2 \sum_{a \in R+} \sum_{p=1}^{\infty} m(g + pa)(g + pa,a)$$

where h is the highest weight of the representation, g is the weight for which the multiplicity is being computed, and d is the sum of the fundamental weights. The first sum on the right-hand side is over all the positive roots, and the inner sum, although formally infinite, in fact only contains a finite number of non-zero terms. It can be shown that $(h + d,h + d) - (g + d,g + d) \neq 0$ for all weights $g < h$ in the weight system. This formula provides a recursive method for computing the weight multiplicities, starting with the known multiplicity $m(h) = 1$, and continuing with weights of

layers 1, 2, 3, etc.

Although this formula provides an algorithm for computing multiplicities, it typically becomes too time-consuming for hand computations beyond rank 5 and dimension 100 and for electronic computation beyond rank 10 and dimension 10000. However, a modified version of this formula, discovered by R. V. Moody and J. Patera, allows the multiplicities to be efficiently calculated (even by hand) for representations of much larger dimension (Moody and Patera 1982). The paper by Moody and Patera presents two important results: first, a method of directly computing only the dominant weights of the representation and none of the other weights, and second, a multiplicity formula derived from the Freudenthal formula by applying the Weyl group to 'compress' it as much as possible. These two results are discussed in the following paragraphs.

The dominant weight algorithm computes the dominant weights according to the following inductive definition of 'depths' of the set of dominant weights in the weight system. We begin with depth(0), consisting of the highest weight h alone. Depth(i) is then defined to be the set of all dominant weights which can be obtained from dominant weights of the previous depth by subtraction of a positive root, and which have not occurred already; i.e.

depth(i) := {Weights g': g' dominant, g' = g - a for some g of depth(i-1), some positive root a} \
depth(i-1),

where \ denotes the set complement. Since there are only finitely many dominant weights, only finitely many of the depths are non-empty; it can be shown (see the following paragraph) that the union of all the depths equals the set of dominant weights in the weight system. Note that at each stage there may be many non-dominant weights which can be obtained by subtraction of a positive root from a dominant weight of the previous depth, but these non-dominant weights are ignored. Thus this algorithm is in principle very similar to the Dynkin weight algorithm with the exceptions that we subtract all positive roots instead of merely simple roots, and only keep the dominant weights generated at each stage. By subtracting all positive roots, and discarding the non-dominant weights, the dominant weights in the weight system can be computed more quickly: every dominant weight lies in a string of dominant weights created by positive root subtractions originating from the highest weight, whereas if we consider only simple-root subtractions (as in the Dynkin algorithm), some dominant weights cannot be reached through strings consisting entirely of dominant weights.

We refer to the set of dominant weights in the weight system of a representation as the 'dominant weight system' of the representation. The proof that the union of all the 'depths' is the dominant weight system is not very difficult.

Lemma (Moody and Patera 1982): Let V be an f.d.i. L-module

with highest weight h. Then for any dominant weight g ($g \neq h$), g is in the dominant weight system of h if and only if there exists a positive root a such that g+a is in the dominant weight system of h.

Proof: Suppose for some dominant weight g and positive root a, g+a is in the dominant weight system of h. Since g is dominant, $(g,b) \geq 0$ for any positive root b (this follows easily from the definitions of positive root and dominant weight). Then $(g+a,a) = (g,a) + (a,a) > 0$ since $(a,a) > 0$. Now the weight string by a through g+a is $(g+a)+qa, \ldots, (g+a)-pa$ where $p,q \geq 0$ and $p-q = 2(g+a,a)/(a,a)$. Hence $p \geq 0$ and $\xi = (g+a)-a$ is in the dominant weight system of h. Conversely, suppose that g is in the dominant weight system of h (and $g \neq h$). We want to show that for some positive root a, g+a is in the dominant weight system of h. By a basic result in representation theory, we know that there exists a positive root b such that g+b is in the weight system of h (see Humphreys (1972), p. 107, esp. Lemma (a)). If g+b is dominant, we are done (i.e. choose a = b). If g+b is not dominant, then $(g+b,a_i) < 0$ for some simple root a_i; hence $2(g+b,a_i)/(a_i,a_i) < 0$, and then considering the weight string by a_i through g+b ($g+b+qa_i, \ldots, g+b-pa_i$, with $p,q \geq 0$ and $p-q = 2(g+b,a_i)/(a_i,a_i)$) we see that $g+b+a_i$ is in the weight system of h. Since g is dominant, $(g,a_i) \geq 0$ and hence $(g+b,a_i) < 0$ gives $(b,a_i) < 0$. Hence $b+a_i$ is a positive root (considering the weight string by a_i through b in the adjoint representation, the weights of which are the

roots of the algebra). We can therefore replace b by $b+a_i$ in the above and repeat. Since the weight system of h is finite, this process must eventually terminate; at the last step we will have a positive root a such that g+a is in the dominant weight system of h.

In order to explain the modified Freudenthal formula we must first introduce some notation. Suppose we wish to compute the multiplicity of a dominant weight g. Let stab(g) denote the subgroup of W stabilizing g, i.e. the set of all w in W such that wg = g. Writing $g := \sum_{i=1}^{n} g_i w_i$, and defining $T := \{i: g_i = 0\}$, it can be shown that stab(g) = $\langle s(a_i): i \text{ in } T\rangle$; i.e. the subgroup of W generated by the reflections $s(a_i)$. We write stab'(g) for the subgroup of GL(V) generated by stab(g) together with the negative of the identity transformation. Then stab'(g) decomposes the root system into orbits $O_1, \ldots O_t$. (The 'orbits' are the equivalence classes of the relation $a \sim b$ if and only if b = wa for some w in stab'(g)). Each orbit O_j, $1 \leq j \leq t$ contains a unique positive root $x_j := \sum_{i=1}^{n} c_{ji} w_i$ satisfying $c_{ji} \geq 0$ for all i in T. (We say that such an x_j is 'subdominant' in the sense that its coefficients relative to the basis of fundamental weights are non-negative on the fundamental weights indexed by T rather than on all fundamental weights.) In the paper of Moody and Patera, and in the documentation of the accompanying programs, these roots are called 'xi-roots'. Writing $[O_j]$ for the size of the orbit O_j, we have the following modified version of the

Freudenthal formula:

$$[(h + d,h + d) - (g + d,g + d)]m(g) = \sum_{j=1}^{t}[O_j]\sum_{p=1}^{\infty} m(g + px_j)(g + px_j,x_j)$$

where as before the inner sum is in fact finite. The core of this thesis is an implementation of this formula in Pascal. A worked example of computation using the modified formula is given in the Appendix.

The superiority of this formula over the Freudenthal formula is especially obvious when the dominant weights in the weight system have on the average many zero coefficients relative to the basis of fundamental weights. If this is the case, then for most dominant weights few of the roots will be xi-roots and the orbit sizes $[O_j]$ will be large, thus allowing a large amount of compression in the formula. In cases where few of the dominant weights have many zero coefficients, the modified formula is not much more efficient than the original formula. It should be pointed out, however, that cases of this latter type do not often arise in practice with larger ranks (say greater than 4) since such representations will usually have extremely high dimensions and very large numbers of dominant weights, and hence computation of the multiplicities is not feasible.

Chapter 3.

Previous implementations of the Freudenthal formula

One of the earliest implementations of the Freudenthal formula is discussed in Agrawala and Belinfante (1969); their programs were written in ALGOL and FORTRAN and were executed on a UNIVAC 1108 at Carnegie-Mellon University. Input to the program consisted of the type and rank of the simple Lie algebra under consideration together with the coefficients of the highest weight of the module for which the weight system and the multiplicities were to be computed. (By the 'coefficients' of the highest weight are meant its coefficients relative to the basis of fundamental weights; these coefficients are sometimes referred to as the 'Dynkin indices'). Output included the complete weight system of the module (both dominant and non-dominant weights) together with the level and multiplicity of each weight. The program first initialized the Cartan matrix and its inverse as well as the scalar products (a_i,a_j) and (w_i,w_j) of the simple roots and fundamental weights. Although the authors do not explicitly mention it, it appears from the ALGOL 60 procedures appended to their paper that integer arithmetic was used exclusively throughout their programs; thus e.g. an integer DETCAR and a matrix CARINV were initialized, while the actual inverse of the Cartan matrix was CARINV / DETCAR. The root system was

computed using the Dynkin weight algorithm. (This is possible since the root system is a weight system, namely that of the adjoint representation.) The weight system of the module was also computed using the Dynkin algorithm. Finally the program calculated the multiplicities of the weights using the Freudenthal formula. The authors found their program time-efficient for simple Lie algebras of ranks up to 8 and irreducible modules of dimensions up to 1000; for a sample of input values satisfying these conditions, their program took an average of 2.5 seconds (UNIVAC 1108) to compute the weight system and the multiplicities.

The paper by Krusemeyer (1971) discusses an implementation of the Freudenthal formula in ALGOL 60 and executed on the ELX8 computer at the University of Utrecht. The author experimented with the program of Agrawala and Belinfante, checking it against previously published tables of weight multiplicities computed (by hand) by H. Freudenthal. Krusemeyer states that "the computing times turned out to be rather large, and in the case of the table for E8, it was impossible to check the results within a few hours", and concludes that the main reason for the unsatisfactory computing times was the large number of weights in the weight systems, especially for high-dimensional representations of E8. In his paper Krusemeyer makes the important observation that since weights conjugate under the Weyl group have the same

multiplicity, and since each weight in the weight system is conjugate to a unique dominant weight, it suffices to compute the multiplicities of the 'possible dominant weights'. (Here a 'possible dominant weight' is any dominant weight g satisfying $(g + d, g + d) < (h + d, h + d)$ where h is the highest weight and d is the sum of the fundamental weights; note that the dominant weights of the representation under consideration are certainly possible dominant weights, but that a possible dominant weight may have a multiplicity of zero.) However, Krusemeyer's program does not include a procedure to compute the possible dominant weights; a list of such weights must form part of the input to the program. His program does include a procedure to calculate the dominant weight equivalent to an arbitrary weight; the theoretical basis of this procedure is the following 'dominant weight lemma'. Let g be a weight, not necessarily dominant, and let l' be the unique dominant weight equivalent to g (i.e. such that g' = wg for some w in the Weyl group). Define weights $g^{(k)}$ recursively as follows:

$g^{(0)} := g$

$g^{(1)} := s(a_i)g$ if $(g,a_i) < 0$ and $(g,a_j) \geq 0$ for $j < i$

$:= g$ if $(g,a_i) \geq 0$ for all i

$g^{(k)} := (g^{(k-1)})^{(1)}$.

Lemma: a) $g = g^{(1)}$ if and only if g is dominant, and b) for some k, $g^{(k)} = g'$.

Proof: a) $g = g^{(1)}$ if and only if $(g,a_j) \geq 0$ for all j. Now

suppose that $g := \sum_{i=1}^{n} g_i w_i$; then $(g,a_j) = (\sum_{i=1}^{n} g_i w_i, a_j) = \sum_{i=1}^{n} g_i (w_i, a_j)$. By definition $2(w_i,a_j)/(a_j,a_j) = \delta_{ij}$, and hence $(w_i,a_j) = \delta_{ij}(a_j,a_j)/2$, giving $(g,a_j) = g_j(a_j,a_j)/2$. Since $(a_j,a_j)/2 > 0$ for all j we conclude that $(g,a_j) \geq 0$ for all j, if and only if $g_j \geq 0$ for all j. b) If $g = g'$ then clearly we can take $n = 0$. So assume that $g \neq g'$; i.e. $(g,a_i) < 0$ for some minimal i. Then we know that $(g + d, g + d) < (g' + d, g' + d)$ where d is the sum of the fundamental weights (see Humphreys 1972, Lemma B, p. 70). Thus we only need to prove that $(g + d, g + d) < (g^{(i)} + d, g^{(i)} + d)$ since for an arbitrary weight f the scalar product $(f + d, f + d)$ may take only finitely many possible values between $(g + d, g + d)$ and $(g' + d, g' + d)$. Now

$$(g^{(i)} + d, g^{(i)} + d) = (s(a_i)g + d, s(a_i)g + d) =$$
$$(s(a_i)g, s(a_i)g) + (d,d) + 2(s(a_i)g, d) =$$
$$(g,g) + (d,d) + 2(g, s(a_i)d).$$

(The reflection $s(a_i)$ is orthogonal, i.e. it preserves the scalar product.) We know that $s(a_i)d = d - a_i$ (see Humphreys 1972, Corollary to Lemma B, p. 50). Then

$$(g^{(i)} + d, g^{(i)} + d) = (g,g) + (d,d) + 2(g, d - a_i) =$$
$$(g,g) + (d,d) + 2(g,d) - 2(g,a_i) =$$
$$(g + d, g + d) - 2(g,a_i) > (g + d, g + d)$$

since (g,a_i) is strictly negative by hypothesis.

A third implementation of the Freudenthal formula was undertaken by Beck and Kolman; see Beck and Kolman (1972) as well as Kolman and Beck (1973a) and (1973b). Their program, written in FORTRAN and executed on an IBM 360/75 at the

University of Pennsylvania, included a subroutine for generating the complete weight system based on the Dynkin algorithm. However, the dominant weights in the weight system were identified as they were computed, and the Freudenthal formula was used only to calculate the multiplicities of the dominant weights. The multiplicities of the other weights were computed using the following result, which the authors refer to as the 'layer lemma'.

Lemma: Let $g := g_1 w_1 + \ldots + g_n w_n$ be a weight in layer k; then $s(a_i)g$ is a weight in layer $k + g_i$.

Proof: Suppose that $g = h - (c_1 a_1 + \ldots + c_n a_n)$ where $c_1 + \ldots c_n = k$; then $s(a_i)g = g - g_i a_i = h - (c_1 a_1 + \ldots + c_n a_n) - g_i a_i$ from which it is clear that $s(a_i)g$ is in layer $c_1 + \ldots + c_n + g_i = k + g_i$.

This result was applied in the program of Beck and Kolman in the following way: if g is a non-dominant weight at level k, then one of its coefficients, say g_i, must be negative. By the layer lemma, $g' := s(a_i)g$ is in layer $k + g_i < k$; since the multiplicities of g and g' are equal and since the multiplicity of g' has already been computed, that of g can be determined immediately. Two output options were allowed in this implementation; one printed the entire weight system with the multiplicity of each weight, and the other printed only the dominant weights and their multiplicities. As an example of time requirements, the program took 90 seconds (IBM 360/75) to compute the weights and multiplicities for a representation of F4 of dimension 226746.

The program described by McKay, Patera, and Sankoff (1977) was written in FORTRAN and executed on a Cyber 74 at the University of Montreal; it computed the 'branching rules' for semisimple Lie algebras of ranks up to 8 and representations of dimensions up to 10000, relative to semisimple subalgebras. (Branching rules are defined as follows. Let L be a semisimple Lie algebra, r an f.d.i. representation of L, and K a semisimple subalgebra of L. The restriction of r to K is in general not irreducible; the branching rule is the decomposition of r as a direct sum of irreducible representations of K.) The computation of the 'restricted' weight system (i.e. the weights with levels greater than or equal to half that of the highest weight), and of the weight multiplicities, is required for the determination of the branching rules. The program of McKay, Patera, and Sankoff used the Dynkin algorithm to compute the weights and the Freudenthal formula for the multiplicities. As an example of time requirements, the authors mention that the computation of the weights and multiplicities for a representation of E7 of dimension 6480 took 13 minutes and 40 seconds (Cyber 74). Although this program introduced no important new computational methods for weight multiplicities, the determination of branching rules is interesting in its own right as a problem which requires computation of the weights and multiplicities of a representation.

Chapter 4.

ONEREPRESENTATION:

An interactive program which computes the multiplicities for a single representation of a given Lie algebra

Introduction

The interactive program ONEREPRESENTATION computes the dominant weights and their multiplicities for a specified representation of a given Lie algebra. This chapter presents an overview of the program, followed by a detailed discussion of its data structures and procedures, a listing of the program, and some examples of its output.

Execution begins with a call to READTYPERANK which asks the user to input the type and rank of the desired Lie algebra. The Dynkin diagram of the algebra is then displayed by DISPLAYDIAGRAM and various initializations are done by MATRIXINITIALIZE, LENGTHINITIALIZE, and ORDERINITIALIZE. The positive roots of the algebra are determined by COMPUTEPOSITIVEROOTS (which calls INSERTROOTS); the roots are stored in a linked list with head pointer ROOTLIST. If desired they will be output by DISPLAYPOSITIVEROOTS. The dominant weights are determined by COMPUTEDOMINANTWEIGHTS (which begins by asking the user to input the highest weight of the representation); the

weights are stored in a linked list with head pointer WEIGHTLIST. They are sorted by decreasing level by SORTDOMINANTWEIGHTS; if desired they will be output by DISPLAYDOMINANTWEIGHTS. COMPUTEMULTIPLICITIES (which calls various other procedures) calculates and displays the multiplicities of the dominant weights together with the total dimension of the representation (computed using the multiplicities of the dominant weights and the sizes of their orbits under the Weyl group). In order to check the calculations, WEYLDIMENSIONFORMULA computes the total dimension of the representation in another way, using Weyl's product formula. The dynamic storage containing the weight system is then recycled, and the user is asked whether to continue with another representation of the same algebra. If the answer is 'yes', control returns to COMPUTEDOMINANTWEIGHTS; otherwise, the dynamic storage containing the root system is recycled, and control returns to READTYPERANK so that the user may input the name of the next algebra. (Execution is terminated by typing the code A0 which is not a valid algebra name.)

The program is in standard Pascal, with one exception: the VALUE segment which is used to initialize the array PRIME. This could easily be replaced by a procedure to compute the primes or to read them from an external file. There is one GOTO statement and one statement label; both occur in COMPUTEPOSITIVEROOTS, and could be removed without requiring major modifications to the code.

All computations are exact; the only numerical data type used is INTEGER. Although rational numbers frequently occur throughout the computation, it is not difficult multiply certain variables by a fixed factor to ensure integrality; for example, the inverse of the Cartan matrix is scaled up by the determinant of the Cartan matrix. The Bourbaki numbering of the simple roots is used throughout the program (Bourbaki 1968, Humphreys 1972). By assumption, the scalar product (a_i,a_i) is 2 for short roots; this causes the scalar product of an arbitrary vector in the root lattice with an arbitrary vector in the weight lattice to be integral. The basis of simple roots is referred to as the 'alpha-basis', and the basis of fundamental weights as the 'omega-basis'.

Documentation of the program

Files

INPUT, OUTPUT

The program is interactive and hence both INPUT and OUTPUT are identified with the user's terminal.

Global constants

MAXRANK

The maximum rank of the Lie algebra. In order to compute multiplicities for representations of algebras of rank greater than MAXRANK, the user should change MAXRANK and recompile the program.

E6ORDER, E7ORDER, E8ORDER, F4ORDER, G2ORDER

The orders of the Weyl groups of algebras E6, E7, E8, F4, and G2.

NUMPRIMES

The first NUMPRIMES prime numbers are stored in array PRIME which is initialized in the VALUE segment. The primes are needed by procedure WEYLDIMENSIONFORMULA.

Global types

NONNEGATIVE = 0..MAXINT

Non-negative integer.

CLASSICALTYPES = (A, BC, D)

This type is used only to define the first index of the array ORDER, which contains the orders of the Weyl groups of the algebras of 'classical' types A, B, C, and D and of ranks from 1 to MAXRANK. B and C are identified in CLASSICALTYPES since the corresponding Weyl groups are isomorphic.

RANKRANGE = 1..MAXRANK

Subrange for array and matrix indices.

VECTOR = ARRAY [RANKRANGE] OF INTEGER

This type is used primarily to define variables containing the coefficients of weights and roots relative to the alpha-basis or omega-basis.

```
ROOTLINK = ROOT;
ROOT =
  RECORD
  ALPHA, OMEGA: VECTOR;
  LEVEL, ORBITSIZE: NONNEGATIVE;
  NEXTROOT: ROOTLINK;
  END;
```

These two types are used to define variables containing data about the positive roots of the algebra. ALPHA and OMEGA

contain respectively the coefficients of a positive root relative to the alpha-basis and omega-basis. The LEVEL of a root is the number of simple roots which must be subtracted from the highest root in order to obtain the root; the levels are needed by the root-generation algorithm (see procedure COMPUTEPOSITIVEROOTS). During the computation of the multiplicity of a weight, if the root is a xi-root (see procedure FINDXIROOTS) then ORBITSIZE will hold the number of roots (not only positive roots) in the orbit of the root under the action of the subgroup of the Weyl group stabilizing the weight extended by the negative of the identity transformation. NEXTROOT is a link to the next root in the list. Although the size of the root system is known in advance (see Humphreys 1972, p. 66, Table 1) it is still more practical to use dynamic storage for the root list, since during the generation of the roots new roots must be inserted into the list between roots already computed. Both the alpha- and omega-basis expressions for the roots are needed: procedure COMPUTEPOSITIVEROOTS uses the omega-basis for generating the roots, and the multiplicity computation uses both the alpha- and omega-basis (see procedure FINDXIROOTS, function DOUBLESUMMATION, and procedure WEYLDIMENSIONFORMULA).

```
WEIGHTLINK = WEIGHT;
WEIGHT =
  RECORD
  OMEGA: VECTOR;
  MULTIPLICITY, DEPTH, LEVEL: NONNEGATIVE;
```

```
  NEXTWEIGHT: WEIGHTLINK;
  END;
```

These two types are used to define variables containing data about the dominant weights of the weight system of a given representation. OMEGA contains the coefficients of a weight relative to the omega-basis. MULTIPLICITY contains the multiplicity of the weight. The DEPTH of the weight is the least number of positive roots which must be subtracted from the highest weight in order to obtain the weight; the depths are needed by the weight-generation algorithm (see procedure COMPUTEDOMINANTWEIGHTS). The LEVEL of the weight is one plus twice the scalar product of the weight with the vector defined by the property that its scalar product with every simple root is 1; the levels have the property that all the dominant weights in the weight system of a representation have levels less than or equal to that of the highest weight of the representation. NEXTWEIGHT is a link to the next weight in the list. Dynamic storage is appropriate for the weight list since it is not known in advance how many dominant weights are in the weight system; the number can be as small as 5 (or smaller) or as large as 80 (or larger).

```
LINK = NODE;
NODE =
  RECORD
  LEFT, RIGHT: LINK;
  NONZERO: BOOLEAN;
  END;
```

These two types are used to define the array CARRIER which contains the data about a weight which is needed to compute

the order of its stabilizer (see function STABILIZERORDER). LEFT and RIGHT are links to the left and right neighbors of the node. Each node attached to CARRIER corresponds to a coefficient of the weight relative to the omega-basis; NONZERO is true if and only if the corresponding coefficient is not zero.

Global variables

CARTANMATRIX, INVERSEMATRIX: ARRAY [RANKRANGE, RANKRANGE] OF INTEGER;

The Cartan matrix of the algebra and its inverse. INVERSEMATRIX is in fact the inverse Cartan matrix multiplied by the determinant of the Cartan matrix; this ensures that all the entries will be integral.

DETERMINANT: NONNEGATIVE;

The determinant of the Cartan matrix.

SQUARELENGTH: VECTOR;

The squares of the lengths of the simple roots, i.e. SQUARELENGTH [i] is the scalar product (a_i, a_i).

LATYPE: CHAR;

The type of the algebra; 'A', 'B', 'C', 'D', 'E', 'F', or 'G'. The type is read directly from the user's terminal into this variable and hence any character value is allowed

in order to avoid a fatal runtime error on input.

RANK: INTEGER;

The rank of the algebra. Since the value is read directly from the user's terminal, any integer is allowed.

ROOTLIST, TEMPROOT: ROOTLINK;

ROOTLIST points to the head node of the list of positive roots. TEMPROOT is a temporary variable used in recycling the root list.

NUMROOTS: NONNEGATIVE;

NUMROOTS is used during the computation of the positive roots to count the number which have already been computed.

WEIGHTLIST, TEMPWEIGHT: WEIGHTLINK;

WEIGHTLIST points to the head node of the list of dominant weights. TEMPWEIGHT is a temporary variable used in recycling the weight list.

NUMWEIGHTS: INTEGER;

NUMWEIGHTS is used during the computation of the dominant weights to count the number which have already been computed.

CARRIER: ARRAY [RANKRANGE] OF LINK;

An array of pointers of type LINK which 'carries' the chain

of records of type NODE created by procedure CREATECHAIN and used by functions AORDER, BCORDER, DORDER, EORDER, FORDER, and STABILIZERORDER to compute the order of the stabilizer of a weight. A global array of pointers is used in order to avoid having to allocate and dispose local dynamic storage each time STABILIZERORDER is called.

ORDER: ARRAY [CLASSICALTYPES, RANKRANGE] OF NONNEGATIVE;
The orders of the algebras of the 'classical' types A, B, C, and D and of ranks from 1 to MAXRANK.

PRIME: ARRAY [1..NUMPRIMES] OF NONNEGATIVE;
The first NUMPRIMES prime numbers. PRIME is initialized by the VALUE segment and is used in procedure WEYLDIMENSIONFORMULA.

Procedures and functions

In the descriptions of the procedures and functions, calls to standard Pascal procedures are not mentioned with the exceptions of NEW and DISPOSE.

FUNCTION POSITIVEANSWER: BOOLEAN;
This funttion, which is called by COMPUTEDOMINANTWEIGHTS and the main block, asks the user for a 'Y' or 'N' (yes or no) answer, reads it, and checks its validity (i.e. checks that the input character is either 'Y' or 'N'). This sequence is

repeated until a valid response has been input, at which point the function returns TRUE for 'Y' and FALSE for 'N'.

PROCEDURE READTYPERANK;

This procedure, which is called by the main block, asks the user for the name (i.e. the type and rank) of an algebra, reads it, and checks its validity (i.e. checks that such a type exists and that the rank is within range). This sequence is repeated until a valid name has been input. Type A1 is not allowed since the weights and multiplicities are well-known (see Humphreys 1972, p. 33, Theorem). Other low-rank algebras are omitted because of isomorphisms among them; see Chapter 1.

PROCEDURE DISPLAYDIAGRAM;

This procedure, which is called by the main block, displays the Dynkin diagram of the algebra. For algebras with two root lengths (i.e. types B, C, F, and G) the arrowhead (< or >) points to the nodes corresponding to the short roots.

PROCEDURE SPECIALLINEARMATRIX;

This procedure, which is called by MATRIXINITIALIZE, initializes CARTANMATRIX for algebras of type A (which are known as special linear algebras, hence the name of the procedure). The initialization for type A has been made a separate procedure since the Cartan matrix for any other algebra can be computed from the type A matrix of the same

rank merely by changing a few entries.

PROCEDURE MATRIXINITIALIZE;
This procedure, which is called by the main block, initializes CARTANMATRIX, DETERMINANT, and INVERSEMATRIX for the algebra of type LATYPE and rank RANK. This procedure calls SPECIALLINEARMATRIX.

PROCEDURE LENGTHINITIALIZE;
This procedure, which is called by the main block, initializes the array SQUARELENGTH.

PROCEDURE ORDERINITIALIZE;
This procedure, which is called by the main block, initializes the array ORDER. It also initializes the array CARRIER: it allocates a record of type NODE for each component of CARRIER. This procedure calls the predefined procedure NEW.

PROCEDURE BASISCHANGE (OMEGA: VECTOR, VAR ALPHA: VECTOR);
This procedure, which is called by INSERTROOT, COMPUTEPOSITIVEROOTS, and COMPUTEMULTIPLICITIES, takes a vector OMEGA represented relative to the omega-basis, and computes the vector ALPHA, the representation of OMEGA relative to the alpha-basis. The conversion is done by multiplying OMEGA by the transpose of INVERSEMATRIX (and hence the components of ALPHA, like those of INVERSEMATRIX,

will be scaled up by a factor of DETERMINANT).

FUNCTION SCALARPRODUCT (VECTOR1, VECTOR2: VECTOR): INTEGER;
This function, which is called by DOUBLESUMMATION and COMPUTEMULTIPLICITIES, computes the scalar product of VECTOR1 with VECTOR2 (by the scalar product is meant the symmetric bilinear form corresponding to the Killing form). One of the vectors must be represented relative to the alpha-basis and one relative to the omega-basis; this allows the scalar product computation to be programmed very simply according to the following argument.
Let $u := \sum_{i=1}^{n} u_i a_i$ and $v := \sum_{j=1}^{n} v_j w_j$. Then $(u,v) = (\sum_{i=1}^{n} u_i a_i , \sum_{j=1}^{n} v_j w_j) = \sum_{i=1}^{n}\sum_{j=1}^{n} (u_i a_i , v_j w_j) = \sum_{i=1}^{n}\sum_{j=1}^{n} u_i v_j (a_i , w_j) = \sum_{i=1}^{n}\sum_{j=1}^{n} u_i v_j \delta_{ij} (a_i , a_i)/2$ (since by the definition of the fundamental weights $2(w_i , a_j)/(a_j , a_j) = \delta_{ij}$. The last expression then equals $\sum_{i=1}^{n} u_i v_i (a_i , a_i)/2$ which is the value computed by SCALARPRODUCT.

FUNCTION EQUALVECTORS (VECTOR1, VECTOR2: VECTOR): BOOLEAN;
This function, which is called by INSERTROOT, COMPUTEDOMINANTWEIGHTS, and PREVIOUSMULTIPLICITY, returns TRUE if and only if VECTOR1 equals VECTOR2.

```
PROCEDURE INSERTROOT (ROOT1: ROOTLINK; NEWROOT: VECTOR;
  NEWLEVEL: NONNEGATIVE);
```

This procedure, which is called by COMPUTEPOSITIVEROOTS, searches for the positive root NEWROOT with level NEWLEVEL in the linked list of positive roots, and inserts it if it is not found, keeping the list ordered by increasing level. Recall that the level of a given root is the number of simple roots (counting repetitions) which must be subtracted from the highest root in order to obtain the given root. Since the level of NEWROOT is therefore one greater than the level of ROOT1^ (the root from which NEWROOT was obtained by subtraction of a simple root), the search for NEWROOT can begin with node ROOT1^. This procedure calls EQUALVECTORS, BASISCHANGE, and the predefined procedure NEW.

PROCEDURE COMPUTEPOSITIVEROOTS;

This procedure, which is called by the main block, initializes the highest root of the algebra and then computes the positive roots. The positive roots are generated as the weights of the adjoint representation: starting with the highest weight of this representation (which is the highest root of the algebra), we subtract simple roots from previously-computed positive roots in order to derive further positive roots. This method generates the positive roots by level, one level at a time. All of the simple roots are at the same level (the deepest level of the positive root system); when we have generated all the simple roots, and attempt to generate further positive roots at the next level, we immediately obtain the

zero root. Thus as soon as the zero root is generated, we know that we have computed all of the positive roots (and none of the other roots). Note that given a positive root, expressed in the basis of fundamental weights, the coefficient of a given fundamental weight is the number of times the corresponding simple root can be subtracted, giving a positive root after each subtraction. This procedure calls BASISCHANGE, INSERTROOT, and the predefined procedure NEW.

PROCEDURE DISPLAYPOSITIVEROOTS;

This procedure, which is called by the main block, displays the list of positive roots, giving both the alpha- and omega-basis representations.

PROCEDURE COMPUTEDOMINANTWEIGHTS;

This procedure, which is called by the main block, implements the dominant weight algorithm of Moody and Patera (1982). The user is asked to input the highest weight of the representation for which the multiplicities are to be computed. (The highest weight is read directly into the head node of the weight list, and the user must confirm the correctness of the weight by typing 'Y' or 'N'; if 'N' is typed, the input loop will repeat.) The procedure then computes the dominant weights with non-zero multiplicities in this representation by the repeated subtraction of positive roots from previously-computed dominant weights;

the weight computation starts by subtracting positive roots from the highest weight. The generation of new dominant weights continues until all the positive roots have been subtracted from all the previously-generated weights. The weight list is kept ordered by increasing depth at all times; recall that the depth of a given weight is the least number of positive roots that must be subtracted from the highest weight in order to obtain the weight. This procedure calls POSITIVEANSWER, EQUALVECTORS, and the predefined procedure NEW.

PROCEDURE SORTDOMINANTWEIGHTS;

This procedure, which is called by the main block, computes the levels of the dominant weights in the weight list and sorts the weights by decreasing level. (The 'level' of a weight g is defined to be $1 + 2(g,d')$ where d' is defined by the condition that $(d',a_i) = 1$ for every simple root a_i.) A simple computational method of determining the scalar product (g,d') is to compute the vector INVERSEROWSUM (containing the sums along the rows of INVERSEMATRIX) and then take the ordinary scalar product of INVERSEROWSUM with the weight. The equality of these two scalar products can be proved as follows. (This result is due to R. V. Moody.)

We denote the inverse of a matrix M by M^{-1}, its transpose by M^t, and its (i,j) entry by M_{ij}. Let A denote

the Cartan matrix. Let $\{a_1, \ldots, a_n\}$ and $\{w_1, \ldots, w_n\}$ be the alpha- and omega-bases. We know (by definition of A) that $A_{ij} = 2(a_i, a_j)/(a_j, a_j)$ and (by definition of d') that $(d', a_i) = 1$ for all i. We know that $w_i = \sum_{j=1}^{n} A^{-1}_{ij} a_j$ (see Humphreys 1972, p. 68); hence given an arbitrary vector $v := \sum_{i=1}^{n} v_i w_i$ expressed relative to the omega-basis, we can write v relative to the alpha-basis as

$$v = \sum_{i=1}^{n} v_i \sum_{j=1}^{n} A^{-1}_{ij} a_j = \sum_{j=1}^{n} \left(\sum_{i=1}^{n} v_i A^{-1}_{ij} \right) a_j .$$

Now since $(d', a_i) = 1$ for all i, we have $d' = \sum_{i=1}^{n} (2/(a_i, a_i)) w_i$ (using the definition of the fundamental weights: $2(w_i, a_j)/(a_j, a_j) = \delta_{ij}$). Then expressing d' relative to the alpha-basis we have

$$d' = \sum_{j=1}^{n} \left(\sum_{i=1}^{n} (2/(a_i, a_i)) A^{-1}_{ij} \right) a_j .$$

Now let $g := \sum_{k=1}^{n} g_k w_k$ be an arbitrary weight. We have

$$(d', g) = \left(\sum_{j=1}^{n} \left(\sum_{i=1}^{n} (2/(a_i, a_i)) A^{-1}_{ij} \right) a_j , \sum_{k=1}^{n} g_k w_k \right) =$$

$$\sum_{j=1}^{n} \left(\sum_{i=1}^{n} (2/(a_i, a_i)) A^{-1}_{ij} \right) g_j ((a_j, a_j)/2)$$

(again using the definition of the fundamental weights).

This reduces to

$$(d';g) = \sum_{j=1}^{n}\Big(\sum_{i=1}^{n}((a_j,a_j)/(a_i,a_i))A^{-1}_{ij}\Big)g_j .$$

Thus it remains to show that for fixed j,

$$\sum_{i=1}^{n}((a_j,a_j)/(a_i,a_i))A^{-1}_{ij} = \sum_{i=1}^{n}A^{-1}_{ji} .$$

Now $A_{ij} = 2(a_i,a_j)/(a_j,a_j)$ and $A_{ji} = 2(a_j,a_i)/(a_i,a_i)$ and hence $A_{ji} = ((a_j,a_j)/(a_i,a_i))A_{ij}$. Denote by Q the diagonal matrix with $Q_{ii} := (a_i,a_i)$. Since $A^t_{ij} = A_{ji}$ we then have $A^t = Q^{-1}AQ$ and hence $(A^t)^{-1} = Q^{-1}A^{-1}Q$. Now $(A^t)^{-1}_{ij} = (A^{-1})^t_{ij} = A^{-1}_{ji}$ and also $(A^t)^{-1}_{ij} = (Q^{-1}A^{-1}Q)_{ij} = ((a_j,a_j)/(a_i,a_i))A^{-1}_{ij}$ and hence we have term-by-term equality between $\sum_{i=1}^{n}((a_j,a_j)/(a_i,a_i))A^{-1}_{ij}$ and $\sum_{i=1}^{n}A^{-1}_{ji}$.

The sorting algorithm used by this procedure is a simple bubble-sort applied to the singly-linked list of weights. This algorithm is not very efficient, but since the weight list is usually not very long (less than 80 weights) and is already nearly sorted (because of the weight-generation algorithm), and since the sorting is done only once for each representation considered, a special effort to ensure efficiency in this procedure did not seem necessary.

PROCEDURE DISPLAYDOMINANTWEIGHTS;

This procedure, which is called by the main block, displays

the list of dominant weights (now sorted by decreasing level) together with the depth and level of each weight.

```
PROCEDURE CREATECHAIN (WEIGHT: VECTOR; VAR FIRST, LAST:
  LINK);
```

This procedure, which is called by STABILIZERORDER, creates a doubly-linked list between the nodes CARRIER [1]^ and CARRIER [RANK]^ with the NONZERO field of each node being TRUE if and only if the corresponding entry of WEIGHT is not zero. FIRST and LAST are set equal to CARRIER [1] and CARRIER [RANK] respectively. The doubly-linked list attached to CARRIER is used by functions AORDER, BCORDER, DORDER, EORDER, FORDER, and STABILIZERORDER.

```
FUNCTION AORDER (VAR L1, L2: LINK): NONNEGATIVE;
FUNCTION BCORDER (VAR L1, L2: LINK): NONNEGATIVE;
FUNCTION DORDER (VAR L1, L2: LINK): NONNEGATIVE;
FUNCTION EORDER (VAR L1, L2: LINK): NONNEGATIVE;
FUNCTION FORDER (VAR L1, L2: LINK): NONNEGATIVE;
```

These functions are all called by STABILIZERORDER; in addition, AORDER is called by the other four, BCORDER is called by FORDER, and DORDER is called by EORDER. Each of these functions analyzes the corresponding type of subchain (i.e. A, B, C, D, E, or F) of the doubly-linked list attached to CARRIER between links L1 and L2, and returns the order of the subgroup of the corresponding Weyl group which stabilizes the weight corresponding to the subchain. (A separate procedure 'GORDER' is not used since the necessary computations for type G are done within STABILIZERORDER.)

FUNCTION STABILIZERORDER (WEIGHT: VECTOR): NONNEGATIVE;

This function, which is called by FINDXIROOTS and COMPUTEMULTIPLICITIES, returns the order of the subgroup of the Weyl group which stabilizes WEIGHT. For types A, B, C, and F, the function simply calls CREATECHAIN to create the doubly-linked list representing WEIGHT and then calls AORDER, BCORDER, or FORDER as appropriate. For types D and E, the function first changes the numbering of the entries of the vector WEIGHT, then calls CREATECHAIN and then DORDER or EORDER as appropriate; the numbering is changed so that the chains created for algebras of these types can be processed in an analogous manner, which is necessary since EORDER may call DORDER (an E-type diagram may have a D-type subdiagram). For algebras of type G, the order of the stabilizer is determined without calling other procedures or functions. As an example, suppose the program is computing multiplicities for algebra E8 and we have a weight (represented relative to the omega-basis) with non-zero coefficients on fundamental weights 1 and 6 and zeroes elsewhere. Considering the Dynkin diagram for type E8 (see p. 12) we see that nodes 1 and 6 split the diagram into two subdiagrams, namely the Dynkin diagrams of algebras D4 and A2. Thus the order of the subgroup of the Weyl group of E8 which stabilizes this weight will be the order of the Weyl group of D4 times the order of the Weyl group of A2. This procedure calls CREATECHAIN, AORDER, BCORDER, DORDER, EORDER, and FORDER. (Functions AORDER, BCORDER, DORDER,

EORDER, FORDER and STABILIZERORDER were originally written by R. Funk and have since been modified by the present author: the global array CARRIER is now used instead of local dynamic storage; the global array ORDER is now used instead of separate global arrays for types A, BC (i.e. B and C) and D; some control structures in small WHILE-DO loops were rewritten; the renumbering in STABILIZERORDER of the weight vectors under cases D and E has been revised; and most variable and function names have been changed.)

PROCEDURE FINDXIROOTS (SUBWEIGHT: VECTOR; SUBGROUPORDER: NONNEGATIVE);

This procedure, which is called by COMPUTEMULTIPLICITIES, determines the xi-roots and their orbit sizes. SUBWEIGHT is a dominant weight in the weight system of the given representation and SUBGROUPORDER is the order of the subgroup of the Weyl group which stabilizes SUBWEIGHT. This subgroup, extended by the negative of the identity transformation, decomposes the complete weight system of the algebra into orbits, each orbit containing a unique positive subdominant root. 'Subdominant' means that the root, when expressed in the omega-basis, has non-negative coefficients where SUBWEIGHT has zero coefficients. These subdominant orbit representatives are called 'xi-roots'. This procedure searches through the positive root list, determining which roots are xi-roots; if a positive root is a xi-root, the ORBITSIZE field of its record is set to the size of its

orbit, otherwise the ORBITSIZE field is set to zero. This procedure calls STABILIZERORDER.

PROCEDURE REFLECTDOMINANT (FACTOR1: VECTOR; VAR DOMINANT1: DOMINANT1: VECTOR);

This procedure, which is called by DOUBLESUMMATION, computes DOMINANT1, the unique dominant weight conjugate to FACTOR1 under the Weyl group. Since the two weights are conjugate, they have the same multiplicity. DOMINANT1 is determined by applying successive reflections to FACTOR1 until the result is dominant: if the coefficient of some fundamental weight is negative, we apply the reflection defined by the corresponding simple root. This process continues until we obtain a dominant weight. The theoretical justification of this algorithm is the 'dominant weight lemma' discussed in Chapter 3.

FUNCTION PREVIOUSMULTIPLICITY (DOMINANT1: VECTOR): NONNEGATIVE;

This function, which is called by DOUBLESUMMATION, searches in the list of dominant weights for the weight DOMINANT1. If DOMINANT1 is found, the function returns its multiplicity (which will already have been computed, by the recursive nature of the modified Freudenthal formula); otherwise, the function returns zero (since if DOMINANT1 is not in the list, it is not a dominant weight of the representation, and hence its multiplicity is zero). This function calls EQUALVECTORS.

FUNCTION DOUBLESUMMATION (SUBWEIGHT: VECTOR): NONNEGATIVE;
This function, which is called by COMPUTEMULTIPLICITIES, computes and returns the value of the double summation in the modified Freudenthal formula. The outer sum is over the orbits of the root system; this sum is implemented by searching through the root list for the xi-roots (the orbit representatives). If a xi-root is encountered, the inner sum is computed; it is a finite sum of terms of the form scalar product times multiplicity. As soon as a zero term is encountered in the inner sum, we know that all further terms are zero. Instead of using the actual multiplicity of FACTOR1, which may not be available (if FACTOR1 is not dominant), we reflect FACTOR1 into the dominant chamber using REFLECTDOMINANT, and then call PREVIOUSMULTIPLICITY to find the multiplicity of the weight DOMINANT1 computed by REFLECTDOMINANT. Note that since DOMINANT1 is a dominant weight of higher level than SUBWEIGHT, its multiplicity will have been computed by the time we call PREVIOUSMULTIPLICITY. This function calls REFLECTDOMINANT, PREVIOUSMULTIPLICITY, and SCALARPRODUCT.

PROCEDURE COMPUTEMULTIPLICITIES;
This procedure, which is called by the main block, computes and outputs the multiplicities of the dominant weights in the weight list. We know that the multiplicity of the highest weight is one, and from this the multiplicities of weights with lower levels are computed using the modified

Freudenthal formula. Note that in order to obtain the correct multiplicity for a given weight, we must divide the value computed by DOUBLESUMMATION by the difference of the two scalar products in local variables HIGHPRODUCT and SUBPRODUCT. The total dimension of the representation is computed (based on the multiplicities of the dominant weights and their orbit sizes) and stored in DIMENSION. This is done so that the computations can be checked by comparing the result obtained by this method with the result given by WEYLDIMENSIONFORMULA. This procedure calls BASISCHANGE, SCALARPRODUCT, STABILIZERORDER, FINDXIROOTS, and DOUBLESUMMATION.

PROCEDURE WEYLDIMENSIONFORMULA (WEIGHT: VECTOR);

This procedure, which is called by the main block, computes the total dimension of the representation with highest weight WEIGHT using the Weyl dimension formula. In order to allow the computations to remain integral, the running product is stored in terms of its prime factorization in the local array POWER. (An entry in the array POWER holds the power of the corresponding prime in the array PRIME.)

Listing of the program

Although the names of constants, types, variables, procedures, and functions appear in upper-case throughout the documentation of the program, the listing of the program is entirely in lower-case for easier reading.

```
program onerepresentation (input, output);

(* the program contains the following procedures and      *)
(* functions (in order of appearance):                    *)
(*                                                        *)
(* function  positiveanswer                               *)
(* procedure readtyperank                                 *)
(* procedure displaydiagram                               *)
(* procedure speciallinearmatrix                          *)
(* procedure matrixinitialize                             *)
(* procedure lengthinitialize                             *)
(* procedure orderinitialize                              *)
(* procedure basischange                                  *)
(* function  scalarproduct                                *)
(* function  equalvectors                                 *)
(* procedure insertroot                                   *)
(* procedure computepositiveroots                         *)
(* procedure displaypositiveroots                         *)
(* procedure computedominantweights                       *)
(* procedure sortdominantweights                          *)
(* procedure displaydominantweights                       *)
(* procedure createchain                                  *)
(* function  aorder                                       *)
(* function  border                                       *)
(* function  dorder                                       *)
(* function  eorder                                       *)
(* function  forder                                       *)
(* function  stabilizerorder                              *)
(* procedure findxiroots                                  *)
(* procedure reflectdominant                              *)
(* function  previousmultiplicity                         *)
(* function  doublesummation                              *)
(* procedure computemultiplicities                        *)
(* procedure weyldimensionformula                         *)
(* onerepresentation (main block)                         *)
```

```
const

(* the maximum rank of the Lie algebra.                          *)

maxrank = 8;

(* the orders of the weyl groups of algebras e6, ..., g2.        *)

e6order = 51840;
e7order = 2903040;
e8order = 696729600;
f4order = 1152;
g2order = 12;

(* the number of primes stored in array 'prime', and             *)
(* listed in the 'value' segment.                                *)

numprimes = 100;
```

```
type

nonnegative = 0..maxint;
classicaltypes = (a, bc, d);
rankrange = 1..maxrank;
vector = array [rankrange] of integer;

(* pointer type and record type used for the positive      *)
(* roots of the algebra. each record of type 'root'        *)
(* contains data about a positive root.                    *)

rootlink = ^root;
root =
  record

  (* the root expressed in the alpha-basis and the         *)
  (* omega-basis.                                          *)

  alpha, omega: vector;

  (* the 'level' of a root is the number of simple roots   *)
  (* which must be subtracted from the highest root in     *)
  (* order to obtain the root.  if the root is a           *)
  (* 'xi-root' (see procedure 'findxiroots'), the          *)
  (* 'orbitsize' is the number of roots (not only          *)
  (* positive roots) in the orbit of the root under the    *)
  (* action of the group generated by the negative of the  *)
  (* identity transformation and the subgroup of the weyl  *)
  (* group which stabilizes the weight for which the       *)
  (* multiplicity is being computed.                       *)

  level, orbitsize: nonnegative;

  nextroot: rootlink;
  end;

(* pointer type and record type used for the dominant      *)
(* weights of the weight system of a given                 *)
(* representation.  each record of type 'weight' contains  *)
(* data about some dominant weight.                        *)

weightlink = ^weight;
weight =
  record

  (* the weight in the omega-basis.                        *)

  omega: vector;

  (* 'multiplicity' is the multiplicity of the weight.     *)
  (* 'depth' is the least number of positive roots which   *)
  (* must be subtracted from the highest weight of the     *)
  (* representation to obtain the weight.  'level' is the  *)
  (* scalar product of the weight with the vector defined  *)
```

```
    (* by the property that its scalar product with every  *)
    (* simple root is 1.                                   *)

    multiplicity, depth, level: nonnegative;

    nextweight: weightlink;
    end;

(* pointer type and record type used by procedure         *)
(* 'createchain' and functions 'aorder' through 'forder'  *)
(* and 'stabilizerorder'.  these procedures compute the   *)
(* order of the subgroup of the weyl group which          *)
(* stabilizes a given weight.  each record contains data  *)
(* about a component of the given weight.                 *)

link = ^node;
node =
  record
  left, right: link;
  nonzero: boolean;
  end;
```

```
var

(* the cartan matrix and its inverse.  'inversematrix' is *)
(* multiplied by the determinant of the cartan matrix so  *)
(* that all the entries will be integral.                 *)

cartanmatrix, inversematrix: array [rankrange, rankrange] of
  integer;

(* the determinant of the cartan matrix.                  *)

determinant: nonnegative;

(* the squares of the lengths of the simple roots.        *)

squarelength: vector;

(* the type of the algebra, 'a' through 'g'.              *)

latype: char;

(* the rank of the algebra.                               *)

rank: integer;

(* 'rootlist' is a pointer to the head node of the list   *)
(* of positive roots. 'temproot' is a temporary variable  *)
(* used in recycling the root list.                       *)

rootlist, temproot: rootlink;

(* the number of roots; used during the computation of    *)
(* the positive roots to count the number which have been *)
(* computed.                                              *)

numroots: nonnegative;

(* 'weightlist' is a pointer to the head node of the list *)
(* of dominant weights in the weight system. 'tempweight' *)
(* is a temporary variable used in recycling the weight   *)
(* list.                                                  *)

weightlist, tempweight: weightlink;

(* the number of weights; used during the computation of  *)
(* the dominant weights to count the number which have    *)
(* been computed.                                         *)

numweights: nonnegative;

(* an array of pointers of type 'link' used to hold the   *)
(* chain created by procedure 'createchain'.              *)

carrier: array [rankrange] of link;
```

```
(* the orders of the weyl groups of the algebras of the   *)
(* classical types a, b, c, and d, and of rank between 1  *)
(* and maxrank.                                           *)

order: array [classicaltypes, rankrange] of nonnegative;

(* an array containing the first 'numprimes' prime        *)
(* numbers; they are needed by procedure                  *)
(* 'weyldimensionformula'.                                *)

prime: array [1..numprimes] of nonnegative;
```

```
value

prime =
  (   2,   3,   5,   7,  11,  13,  17,  19,
     23,  29,  31,  37,  41,  43,  47,  53,
     59,  61,  67,  71,  73,  79,  83,  89,
     97, 101, 103, 107, 109, 113, 127, 131,
    137, 139, 149, 151, 157, 163, 167, 173,
    179, 181, 191, 193, 197, 199, 211, 223,
    227, 229, 233, 239, 241, 251, 257, 263,
    269, 271, 277, 281, 283, 293, 307, 311,
    313, 317, 331, 337, 347, 349, 353, 359,
    367, 373, 379, 383, 389, 397, 401, 409,
    419, 421, 431, 433, 439, 443, 449, 457,
    461, 463, 467, 479, 487, 491, 499, 503,
    509, 521, 523, 541);
```

```
function positiveanswer: boolean;

(* this function asks the user for a yes/no answer, reads *)
(* it, and checks its validity.  this sequence is         *)
(* repeated until a valid answer has been input. the      *)
(* function returns 'true' for 'yes' and 'false' for      *)
(* 'no'.                                                  *)

var

answer: char;

begin (* positiveanswer *)

repeat
  writeln ('y/n: ');
  readln;
  read (answer);
until (answer = 'y') or (answer = 'n');
writeln;
positiveanswer := answer = 'y';

end; (* positiveanswer *)
```

```
procedure readtyperank;

(* this procedure asks the user for the name (i.e. the     *)
(* type and rank) of an algebra, reads it, and checks its *)
(* validity.  this sequence is repeated until a valid     *)
(* name has been input.                                   *)

var

validname: boolean;

begin (* readtyperank *)

writeln ('input the name of the lie algebra.');
repeat
  write ('a2-', maxrank: 1, ', b2-', maxrank: 1, ', c2-');
  write (maxrank: 1, ', d4-', maxrank: 1, ', e6-8, f4, g2');
  writeln ('a0 to stop:');
  readln;
  read (latype, rank);
  if (latype = 'a') and (2 <= rank) and (rank <= maxrank) or
     (latype = 'b') and (2 <= rank) and (rank <= maxrank) or
     (latype = 'c') and (2 <= rank) and (rank <= maxrank) or
     (latype = 'd') and (4 <= rank) and (rank <= maxrank) or
     (latype = 'e') and (6 <= rank) and (rank <= 8) or
     (latype = 'f') and (rank = 4) or
     (latype = 'g') and (rank = 2) or
     (latype = 'a') and (rank = 0)
    then validname := true
    else validname := false;
  writeln;
until validname;

end; (* readtyperank *)
```

```
procedure displaydiagram;

(* this procedure displays the dynkin diagram of the      *)
(* algebra using the bourbaki numbering of the simple     *)
(* roots.  for algebras of types b, c, f, and g the       *)
(* symbols < and > are used to indicate which roots are   *)
(* short and which are long: the short root(s) is (are)   *)
(* on the closed side and the long root(s) is (are) on    *)
(* the open side.                                         *)

var

i: rankrange;

begin (* displaydiagram *)

writeln ('the dynkin diagram (bourbaki numbering)');
writeln;
case latype of
  'a', 'b', 'c', 'd':
    begin
    for i := 1 to rank - 1 do
      write (i: 3);
    case latype of
      'a': writeln (rank: 3);
      'b': writeln ('  >', rank: 3);
      'c': writeln ('  <', rank: 3);
      'd':
        begin
        writeln;
        for i := 1 to rank - 3 do
          write ('   ');
        writeln (rank: 3);
        end;
      end;
    end;
  'e':
    begin
    writeln ('        2');
    write ('  1');
    for i := 3 to rank do
      write (i: 3);
    writeln;
    end;
  'f': writeln ('  1  2  >  3  4');
  'g': writeln ('  1  <  2');
  end;
writeln;

end; (* displaydiagram *)
```

```
procedure speciallinearmatrix;

(* this procedure initializes the cartan matrix for      *)
(* algebras of type a.                                   *)

var

i, j: rankrange;

begin (* speciallinearmatrix *)

for i := 1 to rank do
  for j := 1 to rank do
    cartanmatrix [i, j] := 0;
for i := 1 to rank - 1 do
  begin
  cartanmatrix [i, i] := 2;
  cartanmatrix [i, i+1] := -1;
  cartanmatrix [i+1, i] := -1;
  end;
cartanmatrix [rank, rank] := 2;

end; (* speciallinearmatrix *)
```

```
procedure matrixinitialize;

(* initializes the cartan matrix and its inverse for the   *)
(* algebra of type 'latype' and rank 'rank'.               *)

var

i, j: rankrange;

begin (* matrixinitialize *)

case latype of
  'a':
    begin

    (* initialize the cartan matrix for type a.            *)

    speciallinearmatrix;

    (* initialize the inverse of the cartan matrix for     *)
    (* type a.                                             *)

    determinant := rank + 1;
    for i := 1 to rank do
      begin
      for j := 1 to i do
        inversematrix [i, j] := j * (rank - i + 1);
      for j := i + 1 to rank do
        inversematrix [i, j] := i * (rank - j + 1);
      end;

    end;
  'b':
    begin

    (* initialize the cartan matrix for type b.            *)

    speciallinearmatrix;
    cartanmatrix [rank - 1, rank] := -2;

    (* initialize the inverse of the cartan matrix for     *)
    (* type b.                                             *)

    determinant := 2;
    for i := 1 to rank - 1 do
      begin
      for j := 1 to i do
        inversematrix [i, j] := 2 * j;
      for j := i + 1 to rank do
        inversematrix [i, j] := 2 * i;
      end;
    for j := 1 to rank do
      inversematrix [rank, j] := j;
```

```
    end;
  'c':
    begin

    (* initialize the cartan matrix for type c.          *)

    speciallinearmatrix;
    cartanmatrix [rank, rank - 1] := -2;

    (* initialize the inverse of the cartan matrix for   *)
    (* type c.                                           *)

    determinant := 2;
    for i := 1 to rank do
      begin
      for j := 1 to i do
        inversematrix [i, j] := 2 * j;
      for j := i + 1 to rank - 1 do
        inversematrix [i, j] := 2 * i;
      inversematrix [i, rank] := i;
      end;

    end;
  'd':
    begin

    (* initialize the cartan matrix for type d.          *)

    speciallinearmatrix;
    cartanmatrix [rank - 2, rank] := -1;
    cartanmatrix [rank - 1, rank] :=  0;
    cartanmatrix [rank, rank - 2] := -1;
    cartanmatrix [rank, rank - 1] :=  0;

    (* initialize the inverse of the cartan matrix for   *)
    (* type d.                                           *)

    determinant := 4;
    for i := 1 to rank - 2 do
      begin
      for j := 1 to i do
        inversematrix [i, j] := 4 * j;
      for j := i + 1 to rank do
        inversematrix [i, j] := 4 * i;
      end;
    for i := 1 to rank - 2 do
      begin
      inversematrix [rank - 1, i] := 2 * i;
      inversematrix [rank, i] := 2 * i;
      inversematrix [i, rank - 1] := 2 * i;
      inversematrix [i, rank] := 2 * i;
      end;
    inversematrix [rank - 1, rank - 1] := rank;
    inversematrix [rank, rank] := rank;
```

```
    inversematrix [rank - 1, rank] := rank - 2;
    inversematrix [rank, rank - 1] := rank - 2;

    end;
  'e':
    case rank of
      6: begin

          (* initialize the cartan matrix for algebra e6.  *)

          speciallinearmatrix;
          cartanmatrix [1, 2] := 0;
          cartanmatrix [2, 1] := 0;
          cartanmatrix [2, 3] := 0;
          cartanmatrix [3, 2] := 0;
          cartanmatrix [1, 3] := -1;
          cartanmatrix [3, 1] := -1;
          cartanmatrix [2, 4] := -1;
          cartanmatrix [4, 2] := -1;

          (* initialize the inverse of the cartan matrix  *)
          (* for algebra e6.                              *)

          determinant := 3;
          inversematrix [1, 1] := 4;
          inversematrix [1, 2] := 3;
          inversematrix [1, 3] := 5;
          inversematrix [1, 4] := 6;
          inversematrix [1, 5] := 4;
          inversematrix [1, 6] := 2;
          inversematrix [2, 2] := 6;
          inversematrix [2, 3] := 6;
          inversematrix [2, 4] := 9;
          inversematrix [2, 5] := 6;
          inversematrix [2, 6] := 3;
          inversematrix [3, 3] := 10;
          inversematrix [3, 4] := 12;
          inversematrix [3, 5] := 8;
          inversematrix [3, 6] := 4;
          inversematrix [4, 4] := 18;
          inversematrix [4, 5] := 12;
          inversematrix [4, 6] := 6;
          inversematrix [5, 5] := 10;
          inversematrix [5, 6] := 5;
          inversematrix [6, 6] := 4;
          for i := 2 to 6 do
            for j := 1 to i - 1 do
              inversematrix [i, j] := inversematrix [j, i];

          end;
      7: begin

          (* initialize the cartan matrix for algebra e7.  *)
```

```
        speciallinearmatrix;
        cartanmatrix [1, 2] := 0;
        cartanmatrix [2, 1] := 0;
        cartanmatrix [2, 3] := 0;
        cartanmatrix [3, 2] := 0;
        cartanmatrix [1, 3] := -1;
        cartanmatrix [3, 1] := -1;
        cartanmatrix [2, 4] := -1;
        cartanmatrix [4, 2] := -1;

        (* initialize the inverse of the cartan matrix    *)
        (* for algebra e7.                                *)

        determinant := 2;
        inversematrix [1, 1] := 4;
        inversematrix [1, 2] := 4;
        inversematrix [1, 3] := 6;
        inversematrix [1, 4] := 8;
        inversematrix [1, 5] := 6;
        inversematrix [1, 6] := 4;
        inversematrix [1, 7] := 2;
        inversematrix [2, 2] := 7;
        inversematrix [2, 3] := 8;
        inversematrix [2, 4] := 12;
        inversematrix [2, 5] := 9;
        inversematrix [2, 6] := 6;
        inversematrix [2, 7] := 3;
        inversematrix [3, 3] := 12;
        inversematrix [3, 4] := 16;
        inversematrix [3, 5] := 12;
        inversematrix [3, 6] := 8;
        inversematrix [3, 7] := 4;
        inversematrix [4, 4] := 24;
        inversematrix [4, 5] := 18;
        inversematrix [4, 6] := 12;
        inversematrix [4, 7] := 6;
        inversematrix [5, 5] := 15;
        inversematrix [5, 6] := 10;
        inversematrix [5, 7] := 5;
        inversematrix [6, 6] := 8;
        inversematrix [6, 7] := 4;
        inversematrix [7, 7] := 3;
        for i := 2 to 7 do
          for j := 1 to i - 1 do
            inversematrix [i, j] := inversematrix [j, i];

        end;
    8: begin

        (* initialize the cartan matrix for algebra e8.  *)

        speciallinearmatrix;
        cartanmatrix [1, 2] := 0;
        cartanmatrix [2, 1] := 0;
```

```
        cartanmatrix [2, 3] := 0;
        cartanmatrix [3, 2] := 0;
        cartanmatrix [1, 3] := -1;
        cartanmatrix [3, 1] := -1;
        cartanmatrix [2, 4] := -1;
        cartanmatrix [4, 2] := -1;

        (* initialize the inverse of the cartan matrix    *)
        (* for algebra e8.                                *)

        determinant := 1;
        inversematrix [1, 1] := 4;
        inversematrix [1, 2] := 5;
        inversematrix [1, 3] := 7;
        inversematrix [1, 4] := 10;
        inversematrix [1, 5] := 8;
        inversematrix [1, 6] := 6;
        inversematrix [1, 7] := 4;
        inversematrix [1, 8] := 2;
        inversematrix [2, 2] := 8;
        inversematrix [2, 3] := 10;
        inversematrix [2, 4] := 15;
        inversematrix [2, 5] := 12;
        inversematrix [2, 6] := 9;
        inversematrix [2, 7] := 6;
        inversematrix [2, 8] := 3;
        inversematrix [3, 3] := 14;
        inversematrix [3, 4] := 20;
        inversematrix [3, 5] := 16;
        inversematrix [3, 6] := 12;
        inversematrix [3, 7] := 8;
        inversematrix [3, 8] := 4;
        inversematrix [4, 4] := 30;
        inversematrix [4, 5] := 24;
        inversematrix [4, 6] := 18;
        inversematrix [4, 7] := 12;
        inversematrix [4, 8] := 6;
        inversematrix [5, 5] := 20;
        inversematrix [5, 6] := 15;
        inversematrix [5, 7] := 10;
        inversematrix [5, 8] := 5;
        inversematrix [6, 6] := 12;
        inversematrix [6, 7] := 8;
        inversematrix [6, 8] := 4;
        inversematrix [7, 7] := 6;
        inversematrix [7, 8] := 3;
        inversematrix [8, 8] := 2;
        for i := 2 to 8 do
          for j := 1 to i - 1 do
            inversematrix [i, j] := inversematrix [j, i];

      end;
    end;
'f':
```

```
      begin

      (* initialize the cartan matrix for type f.                    *)

      speciallinearmatrix;
      cartanmatrix [2, 3] := -2;

      (* initialize the inverse of the cartan matrix for             *)
      (* type f.                                                     *)

      determinant := 1;
      inversematrix [1, 1] := 2;
      inversematrix [1, 2] := 3;
      inversematrix [1, 3] := 4;
      inversematrix [1, 4] := 2;
      inversematrix [2, 1] := 3;
      inversematrix [2, 2] := 6;
      inversematrix [2, 3] := 8;
      inversematrix [2, 4] := 4;
      inversematrix [3, 1] := 2;
      inversematrix [3, 2] := 4;
      inversematrix [3, 3] := 6;
      inversematrix [3, 4] := 3;
      inversematrix [4, 1] := 1;
      inversematrix [4, 2] := 2;
      inversematrix [4, 3] := 3;
      inversematrix [4, 4] := 2;

      end;
    'g':
      begin

      (* initialize the cartan matrix for type g.                    *)

      cartanmatrix [1, 1] :=  2;
      cartanmatrix [1, 2] := -1;
      cartanmatrix [2, 1] := -3;
      cartanmatrix [2, 2] := 2;

      (* initialize the inverse of the cartan matrix for             *)
      (* type g.                                                     *)

      determinant := 1;
      inversematrix [1, 1] := 2;
      inversematrix [1, 2] := 1;
      inversematrix [2, 1] := 3;
      inversematrix [2, 2] := 2;

      end;
    end;

  end; (* matrixinitialize *)
```

```
procedure lengthinitialize;

(* this procedure initializes the array 'squarelength';    *)
(* 'squarelength [i]' contains the square of the length    *)
(* of simple root  i.  we make the convention that the     *)
(* squares of the lengths of the short roots are 2;  this  *)
(* causes the scalar product of an arbitrary vector in     *)
(* the root lattice with an arbitrary vector in the        *)
(* weight lattice to be integral.                          *)

var

i: rankrange;

begin (* lengthinitialize *)

if latype = 'b'
  then
    begin
    for i := 1 to rank - 1 do
      squarelength [i] := 4;
    squarelength [rank] := 2;
    end
  else
    begin
    for i := 1 to rank do
      squarelength [i] := 2;
    if latype = 'c'
      then squarelength [rank] := 4;
    if latype = 'f'
      then
        begin
        squarelength [1] := 4;
        squarelength [2] := 4;
        end;
    if latype = 'g'
      then squarelength [2] := 6;
    end;

end; (* lengthinitialize *)
```

```
procedure orderinitialize;

  (* this procedure initializes the array 'order' (which    *)
  (* contains the orders of the weyl groups of the          *)
  (* classical algebras (types a, b, c, and d)) and the     *)
  (* vector 'carrier' used by function 'stabilizerorder'.   *)

  var

  i: rankrange;

  begin (* orderinitialize *)

  (* initialize the array 'order'.                          *)

  order [a, 1] := 2;
  order [bc, 1] := 2;
  order [d, 1] := 1;
  for i := 2 to maxrank do
    begin
    order [a, i] := (i + 1) * order [a, i - 1];
    order [bc, i] := 2 * i * order [bc, i - 1];
    order [d, i] := 2 * i * order [d, i - 1];
    end;

  (* initialize the carrier.                                *)

  for i := 1 to maxrank do
    new (carrier [i]);

  end; (* orderinitialize *)
```

```
procedure basischange (omega: vector; var alpha: vector);

(* given a vector 'omega' represented in the omega-basis  *)
(* (the basis of fundamental dominant weights), this      *)
(* procedure computes the vector 'alpha', the             *)
(* representation of 'omega' in the alpha-basis (the      *)
(* basis of simple roots).  the conversion is done by     *)
(* multiplying 'omega' by the transpose of                *)
(* 'inversematrix'.  the components of 'alpha' are in     *)
(* fact scaled up by a factor of 'determinant'.           *)

var

i, j: rankrange;

begin (* basischange *)

for i := 1 to rank do
  begin
  alpha [i] := 0;
  for j := 1 to rank do
    alpha [i] :=
      alpha [i] + inversematrix [j, i] * omega [j];
  end;

end; (* basischange *)
```

```
function scalarproduct (vector1, vector2: vector): integer;

(* this procedure computes the scalar product of the      *)
(* vectors 'vector1' and 'vector2'; one must be           *)
(* represented in the omega-basis and the other in the    *)
(* alpha-basis .                                          *)

var

i: rankrange;
sum: integer;

begin (* scalarproduct *)

sum := 0;
for i := 1 to rank do
  sum := sum + vector1 [i] * vector2 [i] * squarelength [i];
scalarproduct := sum div 2;

end; (* scalarproduct *)
```

```
function equalvectors (vector1, vector2: vector): boolean;

(* this function returns 'true' iff 'vector1' equals      *)
(* 'vector2'.                                             *)

var

i: rankrange;
result: boolean;

begin (* equalvectors *)

result := true;
for i := 1 to rank do
  result := result and (vector1 [i] = vector2 [i]);
equalvectors := result;

end; (* equalvectors *)
```

```
procedure insertroot (root1: rootlink; newroot: vector;
  newlevel: nonnegative);

(* this procedure searches for the positive root          *)
(* 'newroot' with level 'newlevel' in the list of         *)
(* positive roots, and inserts it if it is not found,     *)
(* keeping the list ordered by increasing level.  by the  *)
(* level of a given positive root is meant the number of  *)
(* simple roots which must be subtracted from the highest *)
(* root in order to obtain the given root.  since the     *)
(* level of 'newroot' is therefore one greater than the   *)
(* level of the root from which it was obtained by        *)
(* subtraction of a simple root, the search for 'newroot' *)
(* can begin with node 'root1^' (the node containing the  *)
(* root from which 'newroot' was obtained.                *)

var

root2, root3: rootlink;
insert: boolean;

begin (* insertroot *)

(* initialize root pointers for search.                   *)

root2 := root1;
root3 := root1^. nextroot;

(* the following loop searches through the root list for  *)
(* 'newroot'.  after the loop has terminated, 'insert'    *)
(* will be true iff 'newroot' has not been found in the   *)
(* root list.                                             *)

insert := true;
while root3 <> nil do
  if root3^. level < newlevel
    then
      begin
      root2 := root3;
      root3 := root3^. nextroot;
      end
    else
      if root3^. level <> newlevel
        then
          root3 := nil
        else
          if equalvectors (root3^. omega, newroot)
            then
              begin
              insert := false;
              root3 := nil;
              end
            else
              begin
```

```
                root2 := root3;
                root3 := root3^. nextroot;
                end;

(* insert 'newroot' if necessary.                                    *)

if insert
  then
    begin
    new (root3);
    root3^. nextroot := root2^. nextroot;
    root2^. nextroot := root3;
    root3^. level := newlevel;
    root3^. omega := newroot;
    basischange (root3^. omega, root3^. alpha);
    numroots := numroots + 1;
    end;

end; (* insertroot *)
```

```
procedure computepositiveroots;

(* this procedure computes the positive roots of the      *)
(* algebra.  they are generated as the weights of the     *)
(* adjoint representation: starting with the highest      *)
(* weight of this representation (which is the highest    *)
(* root of the algebra), we subtract simple roots from    *)
(* previously-computed positive roots in order to derive  *)
(* further positive roots.  this method generates the     *)
(* positive roots by level, one level at a time.  all of  *)
(* the simple roots are at the same level (the deepest    *)
(* level of the positive root system); when we have       *)
(* generated all the simple roots, and attempt to         *)
(* generate further positive roots at the next level, we  *)
(* immediately obtain the zero root.  thus as soon as the *)
(* zero root is generated, we know that we have computed  *)
(* all of the positive roots (and none of the other       *)
(* roots).  note that given a positive root, expressed in *)
(* the basis of fundamental dominant weights, the         *)
(* coefficient of a given fundamental weight is the       *)
(* number of times the corresponding simple root can be   *)
(* subtracted, giving a positive root after each          *)
(* subtraction.                                           *)

label
  0;

var

  i, j: rankrange;
  count: nonnegative;
  root1: rootlink;
  newroot: vector;
  zerovector: boolean;

begin (* computepositiveroots *)

(* store the highest root in the head node of the root    *)
(* list.                                                  *)

  new (rootlist);
  rootlist^. level := 0;
  for i := 1 to rank do
    rootlist^. omega [i] := 0;
  case latype of
    'a':
      begin
      rootlist^. omega [1] := 1;
      rootlist^. omega [rank] := 1;
      end;
    'b':
      if rank = 2
        then rootlist^. omega [2] := 2
```

```
      else rootlist^. omega [2] := 1;
    'c':
      rootlist^. omega [1] := 2;
    'd':
      begin
      rootlist^. omega [2] := 1;
      if rank = 3
        then rootlist^. omega [3] := 1;
      end;
    'e':
      case rank of
        6: rootlist^. omega [2] := 1;
        7: rootlist^. omega [1] := 1;
        8: rootlist^. omega [8] := 1;
        end;
    'f':
      rootlist^. omega [1] := 1;
    'g':
      rootlist^. omega [2] := 1;
    end;
  rootlist^. nextroot := nil;
  basischange (rootlist^. omega, rootlist^. alpha);
  numroots := 1;

  (* the following segment computes the positive roots by   *)
  (* repeated subtraction of simple roots from              *)
  (* previously-computed positive roots.  it starts by      *)
  (* subtracting simple roots from the highest root.  as    *)
  (* soon as this algorithm generates the zero vector, all  *)
  (* the postive roots have been computed.                  *)

  root1 := rootlist;
  while root1 <> nil do
    begin
    for i := 1 to rank do

      (* see if we can subtract simple root  i from this    *)
      (* positive root.                                     *)

      if root1^. omega [i] > 0
        then
          begin

          (* store this positive root in 'newroot'.         *)

          newroot := root1^. omega;

          (* subtract simple root  i and test for the zero  *)
          (* root.                                          *)

          for count := 1 to root1^. omega [i] do
            begin
            for j := 1 to rank do
              newroot [j] :=
```

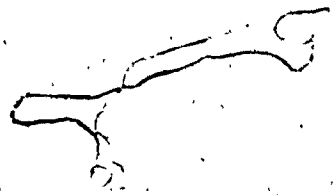
```
            newroot [j] - cartanmatrix [i, j];
          zerovector := true;
          for j := 1 to rank do
            zerovector := zerovector and (newroot [j] = 0);
          if zerovector
           then
             (* the zero vector has been generated;      *)
             (* escape.                                  *)
             goto 0
           else
             (* the new root is not zero; store it in    *)
             (* the root list.                           *)
             insertroot (root1, newroot, root1. level +
               count);
          end;

      end;
  root1 := root1^. nextroot;
  end;

0:;

end; (* computepositiveroots *)
```

```
procedure displaypositiveroots;

(* this procedure displays the list of positive roots      *)
(* (ordered by level), giving both the alpha-basis and     *)
(* omega-basis representations.                            *)

var

root1: rootlink;
i: rankrange;
count: nonnegative;

begin (* displaypositiveroots *)

(* output the headings.                                    *)

write ('    ');
for i := 1 to rank - 1 do
  write ('    ');
write (' alpha');
for i := 1 to rank - 1 do
  write ('    ');
writeln (' omega');

(* output the roots.                                       *)

count := 1;
root1 := rootlist;
while root1 <> nil do
  begin
  write (count: 3, '    ');
  for i := 1 to rank do
    write (root1^. alpha [i] div determinant: 3);
  write ('    ');
  for i := 1 to rank do
    write (root1^. omega [i]: 3);
  writeln;
  root1 := root1^. nextroot;
  count := count + 1;
  end;
writeln;

end; (* displaypositiveroots *)
```

```
procedure computedominantweights;

(* this procedure reads in the highest weight of the       *)
(* representation for which the multiplicities are to be   *)
(* computed.  it then computes the dominant weights with   *)
(* non-zero weight spaces in this representation and       *)
(* stores them, ordered by increasing depth, in the list   *)
(* with head pointer 'weightlist'.  the depth of a given   *)
(* weight is the least number of positive roots which      *)
(* must be subtracted from the highest weight in order to  *)
(* obtain the given weight.                                *)

var

dominant, found: boolean;
root1: rootlink;
newweight: vector;
newdepth: nonnegative;
weight1, weight2, weight3: weightlink;
i: rankrange;

begin (* computedominantweights *)

(* read the highest weight into the head node of the       *)
(* weight list, repeating until the user has confirmed     *)
(* its correctness; then set its depth field to zero.      *)

new (weightlist);
repeat
  writeln ('input the highest weight in the basis of');
  writeln ('fundamental dominant weights (bourbaki');
  writeln ('numbering) on one line.');
  readln;
  for i := 1 to rank do
    read (weightlist^. omega [i]);
  writeln;
  writeln ('is it correct ?');
until positiveanswer;
weightlist^. depth := 0;
weightlist^. nextweight := nil;
numweights := 1;

(* the following segment computes the dominant weights by  *)
(* repeated subtraction of positive roots from             *)
(* previously-computed dominant weights.  it starts by     *)
(* subtracting positive roots from the highest weight.     *)
(* the generation of new potential weights continues       *)
(* until all the positive roots have been subtracted from  *)
(* all the previously-computed weights.  a newly-          *)
(* generated potential weight is an actual weight iff it   *)
(* is dominant.                                            *)

weight1 := weightlist;
while weight1 <> nil do
```

```
begin
root1 := rootlist;
while root1 <> nil do
  begin

  (* subtract the positive root from the dominant       *)
  (* weight.                                            *)

  for i := 1 to rank do
    newweight [i] :=
      weight1^. omega [i] - root1^. omega [i];

  (* check to see if the potential weight 'newweight'   *)
  (* is dominant.                                       *)

  dominant := true;
  for i := 1 to rank do
    dominant := dominant and (newweight [i] >= 0);
  if dominant
    then

      (* 'newweight' is dominant and hence is a weight; *)
      (* search for it in the weight list and insert it *)
      (* if it is not found.                            *)

      begin
      newdepth := weight1^. depth + 1;
      found := false;
      weight2 := weightlist;
      weight3 := weightlist^. nextweight;
      while weight3 <> nil do
        if equalvectors (weight3^. omega, newweight)
          then
            begin
            found := true;
            weight3 := nil;
            end
          else
            if newdepth < weight3^. depth
              then weight3 := nil
              else
                begin
                weight2 := weight3;
                weight3 := weight3^. nextweight
                end;
      if not found
        then
          begin
          new (weight3);
          weight3^. omega := newweight;
          weight3^. depth := newdepth;
          weight3^. nextweight := weight2^. nextweight;
          weight2^. nextweight := weight3;
          numweights := numweights + 1;
```

```
                end;
            end;

        root1 := root1^.nextroot;
        end;
    weight1 := weight1^.nextweight;
    end;

end; (* computedominantweights *)
```



```
procedure sortdominantweights;

(* this procedure computes the levels of the dominant     *)
(* weights in the weight list, and sorts the weights by   *)
(* decreasing level.  formally, the level of a weight is  *)
(* defined to be one plus twice the scalar product of the *)
(* weight with the vector 'rhohat', where 'rhohat' is     *)
(* defined by the condition that its scalar product with  *)
(* any simple root is 1.  a simple computational method   *)
(* of determining this scalar product is to compute the   *)
(* vector 'inverserowsum' (containing the sums along the  *)
(* rows of the inverse cartan matrix) and then take the   *)
(* ordinary scalar product of the weight with             *)
(* 'inverserowsum'.                                       *)

var

inverserowsum: vector;
i, j: rankrange;
weight1, weight2, weight3: weightlink;
done: boolean;

begin (* sortdominantweights *)

(* compute 'inverserowsum'.                               *)

for i := 1 to rank do
  begin
  inverserowsum [i] := 0;
  for j := 1 to rank do
    inverserowsum [i] :=
      inverserowsum [i] + inversematrix [i, j];
  end;

(* compute the levels of the weights in the weight list.  *)

weight1 := weightlist;
while weight1 <> nil do
  begin
  weight1^. level := 0;
  for i := 1 to rank do
    weight1^. level := weight1^. level +
      inverserowsum [i] * weight1^. omega [i];
  weight1^. level :=
    (2 * weight1^. level) div determinant + 1;
  weight1 := weight1^. nextweight;
  end;

(* sort the weights by decreasing level.                  *)

repeat
  done := true;
  weight1 := nil;
  weight2 := weightlist;
```

```
    while weight2 <> nil do
      begin
      weight3 := weight2^. nextweight;
      if weight3 <> nil
        then
          if weight2^.level < weight3^.level
            then
              begin
              done := false;
              weight2^. nextweight := weight3^. nextweight;
              weight3^. nextweight := weight2;
              if weight1 <> nil
                then weight1^. nextweight := weight3
                else weightlist := weight3;
              weight2 := weight3;
              weight3 := weight3^. nextweight;
              end;
      weight1 := weight2;
      weight2 := weight2^. nextweight;
      end;
  until done;

  end; (* sortdominantweights *)
```

```
procedure displaydominantweights;

(* this procedure displays the list of weights together  *)
(* with the depth and level of each weight.              *)

var

i: rankrange;
count: nonnegative;
weight1: weightlink;

begin (* displaydominantweights *)

(* output the titles.                                    *)

write ('       depth    level');
for i := 1 to rank - 1 do
  write ('   ');
writeln ('weight');

(* output the weights.                                   *)

weight1 := weightlist;
count := 1;
while weight1 <> nil do
  begin
  write (count: 3, '   ', weight1^. depth: 6, '   ');
  write (weight1^. level: 6, '   ');
  for i := 1 to rank do
    write (weight1^. omega [i]: 3);
  writeln;
  weight1 := weight1^. nextweight;
  count := count + 1;
  end;
writeln;

end; (* displaydominantweights *)
```

```
procedure createchain (weight: vector; var first,
  last: link);

(* this procedure creates a doubly-linked list between    *)
(* the nodes 'carrier [1]^' and 'carrier [rank]' with     *)
(* the 'nonzero' field of each node being 'true' iff the  *)
(* corresponding entry of 'weight' is non-zero, and       *)
(* returns the values 'carrier [1]' for 'first' and       *)
(* 'carrier [rank]' for 'last'.                           *)

var

i: rankrange;

begin (* createchain *)

for i := 1 to rank do
  carrier [i]^. nonzero := weight [i] <> 0;
for i := rank + 1 to maxrank do
  carrier [i]^. nonzero := false;
for i := 1 to rank - 1 do
  carrier [i]^. right := carrier [i + 1];
for i := rank to maxrank do
  carrier [i]^. right := nil;
carrier [1]^. left := nil;
for i := 2 to rank do
  carrier [i]^. left := carrier [i - 1];
for i := rank + 1 to maxrank do
  carrier [i]^. left := nil;
first := carrier [1];
last := carrier [rank];

end; (* createchain *)
```

```
function aorder (var l1, l2: link): nonnegative;

(* this function analyzes the a-type subchain between      *)
(* links 'l1' and 'l2' and returns the order of the        *)
(* subgroup of the weyl group which stabilizes this        *)
(* subchain.                                               *)

var

result, count: nonnegative;
continue: boolean;

begin (* aorder *)

result := 1;
repeat
  continue := l1 <> nil;
  while continue do
    if l1^. nonzero
      then
        begin
        l1 := l1^. right;
        continue := l1 <> nil;
        end
      else continue := false;
  count := 0;
  continue := l1 <> nil;
  while continue do
    if not l1^. nonzero
      then
        begin
        count := count + 1;
        l1 := l1^. right;
        continue := l1 <> nil;
        end
      else continue := false;
  if count <> 0
    then result := result * order [a, count];
until l1 = nil;
aorder := result;

end; (* aorder *)
```

```
function bcorder (var l1, l2: link): nonnegative;

(* this function analyzes the b- or c-type subchain        *)
(* between links 'l1' and 'l2' and returns the order of    *)
(* the subgroup of the weyl group which stabilizes this    *)
(* subchain.                                               *)

var

l3: link;
continue: boolean;
result, count: nonnegative;

begin (* bcorder *)

if l2^. nonzero
  then
    begin
    l2^. left^. right := nil;
    result := aorder (l1, l2^. left);
    end
  else
    begin
      l3 := l2^. left;
      if l3^. nonzero
        then
          if l3^. left <> nil
            then
              begin
              l3^. left^. right := nil;
              result :=
                aorder (l1, l3^. left) * order [a, 1];
              end
            else result := order [a, 1]
        else
          begin
          count := 0;
          l3 := l3^. left;
          continue := l3 <> nil;
          while continue do
            if l3^. nonzero
              then continue := false
              else
                begin
                count := count + 1;
                l3 := l3^. left;
                continue := l3 <> nil;
                end;
          if l3 <> nil
            then
              begin
              l3^. right := nil;
              result :=
                aorder (l1, l3) * order [bc, 2 + count];
```

```
                end
              else result := order [bc, 2 + count];
            end;
        end;
    bcorder := result;

end; (* bcorder *)
```

```
function dorder (var l1, l2: link): nonnegative;

(* this function analyzes the d-type subchain between      *)
(* links 'l1' and 'l2' and returns the order of the        *)
(* subgroup of the weyl group which stabilizes this        *)
(* subchain.                                               *)

var

l3: link;
continue: boolean;
result, count: nonnegative;

begin (* dorder *)

if l1^. nonzero
  then result := aorder (l1^. right, l2)
  else
    begin
    l3 := l1^. right;
    if l3^. nonzero
      then
        begin
        l1^. right := l3^. right;
        l3^. right^. left := l1;
        result := aorder (l1, l2);
        end
      else
        begin
        l3 := l3^. right;
        if l3^. nonzero
          then
            begin
            l3^. right^. left := nil;
            result := order [a, 1] * order [a, 1] *
              aorder (l3^. right, l2);
            end
          else
            begin
            l3 := l3^. right;
            if l3^. nonzero
              then
                begin
                if l3 <> l2
                  then result := order [a, 3] *
                                 aorder (l3^. right, l2)
                  else result := order [a, 3];
                end
              else
                begin
                count := 1;
                l3 := l3^. right;
                continue := l3 <> nil;
                while continue do
```

```
                    if l3^. nonzero
                      then continue := false
                      else
                        begin
                        l3 := l3^. right;
                        continue := l3 <> nil;
                        count := count + 1;
                        end;
                    result := order [d, count + 3];
                    if l3 <> nil
                      then if l3^. right <> nil
                             then result := result *
                                    aorder (l3^. right, l2);
                  end;
                end;
            end;
        end;
    dorder := result;

    end; (* dorder *)
```

```
function eorder (var l1, l2: link): nonnegative;

(* this function analyzes the e-type subchain between      *)
(* links 'l1' and 'l2' and returns the order of the        *)
(* subgroup of the weyl group which stabilizes this        *)
(* subchain.                                               *)

var

result, count: nonnegative;
done: boolean;
l3: link;

begin (* eorder *)

if l1^. nonzero
  then result := dorder (l1^. right, l2)
  else
    begin
    l3 := l1^. right;
    if l3^. nonzero
      then
        begin
        l3^. right^. left := nil;
        result := order [a, 1] * aorder (l3^. right, l2);
        end
      else
        begin
        l3 := l3^. right;
        if l3^. nonzero
          then
            begin
            l1^. right^. right := l3^. right;
            l3^. right^. left := l1^. right;
            result := aorder (l1, l2);
            end
          else
            begin
            done := false;
            count := 0;
            repeat
              count := count + 1;
              l3 := l3^. right;
              if l3^. nonzero
                then
                  begin
                  result := aorder (l3^. right, l2);
                  case count of
                    1: result := order [a, 1] *
                         order [a, 2] * result;
                    2: result := order [a, 4] * result;
                    3: result := order [d, 5] * result;
                    4: result := e6order * result;
                    5: result := e7order * result;
```

```
                end;
              done := true;
              end;
        until (13^. right = nil) or done;
        if not done
          then
            case rank of
              6: result := e6order;
              7: result := e7order;
              8: result := e8order;
              end;
        end;
    end;
end;
eorder := result;

end; (* eorder *)
```

```
function forder (var l1, l2: link): nonnegative;

(* this function analyzes the f-type subchain between      *)
(* links 'l1' and 'l2' and returns the order of the        *)
(* subgroup of the weyl group which stabilizes this        *)
(* subchain.                                               *)

var

result: nonnegative;

begin (* forder *)

if l2^. nonzero
  then result := bcorder (l1, l2^. left)
  else
    begin
    l2 := l2^. left;
    if l2^. nonzero
      then
        begin
        l2^. left^. right := nil;
        result := order [a, 1] * aorder (l1, l2. left);
        end
      else
        begin
        l2 := l2^. left;
        if l2^. nonzero
          then
            if l2^. left^. nonzero
              then result := order [a, 2]
              else result := order [a, 1] * order [a, 2]
          else
            if l2^. left^. nonzero
              then result := order [bc, 3]
              else result := f4order;
        end;
    end;
forder := result;

end; (* forder *)
```

```
function stabilizerorder (weight: vector): nonnegative;

(* this function returns the order of the subgroup of the *)
(* weyl group which stabilizes 'weight'.  for types a, b, *)
(* c, and f, function 'stabilizerorder' simply calls      *)
(* procedure 'createchain' to create the doubly-linked    *)
(* list representing 'weight' and then calls function     *)
(* 'aorder', 'bcorder', or 'forder' as appropriate.  for  *)
(* types d and e, function 'stabilizerorder' first        *)
(* changes the numbering of the entries of the vector     *)
(* 'weight', then calls 'createchain' and then 'dorder'   *)
(* or 'eorder' as appropriate; the numbering is changed   *)
(* so that the chains created for algebras of these types *)
(* can be processed in an analogous manner, which is      *)
(* necessary since 'eorder' may call 'dorder' (an e-type  *)
(* diagram may have a d-type subdiagram). for algebras of *)
(* type g, the order of the stabilizer is determined      *)
(* without calling other procedures or functions.         *)

var

i: rankrange;
temp: integer;
first, last: link;

begin (* stabilizerorder *)

case latype of
  'a':
    begin
    createchain (weight, first, last);
    stabilizerorder := aorder (first, last);
    end;
  'b', 'c':
    begin
    createchain (weight, first, last);
    stabilizerorder := bcorder (first, last);
    end;
  'd':
    begin
    for i := 1 to rank div 2 do
      begin
      temp := weight [i];
      weight [i] := weight [rank + 1 - i];
      weight [rank + 1 - i] := temp;
      end;
    createchain (weight, first, last);
    stabilizerorder := dorder (first, last);
    end;
  'e':
    begin
    temp := weight [2];
    weight [2] := weight [3];
    weight [3] := temp;
```

```
      createchain (weight, first, last);
      stabilizerorder := eorder (first, last);
      end;
    'f':
      begin
      createchain (weight, first, last);
      stabilizerorder := forder (first, last);
      end;
    'g':
      if weight [1] = 0
        then
          if weight [2] = 0
            then stabilizerorder := g2order
            else stabilizerorder := order [a, 1]
        else
          if weight [2] = 0
            then stabilizerorder := order [a, 1]
            else stabilizerorder := 1;
  end;

end; (* stabilizerorder *)
```

```
procedure findxiroots (subweight: vector; subgroupbrder:
 nonnegative);

(* 'subweight' is a dominant weight of the highest weight *)
(* and 'subgrouporder' is the order of the subgroup of    *)
(* the weyl group which stabilizes 'subweight'. the       *)
(* group generated by this subgroup and the negative of   *)
(* the identity transformation decomposes the complete    *)
(* root system of the algebra into orbits, each orbit     *)
(* containing a unique subdominant root. a root is        *)
(* called 'subdominant' if, when expressed in the         *)
(* omega-basis, it has nonnegative coefficients where     *)
(* 'subweight' has zero coefficients. these subdominant   *)
(* orbit representatives are called 'xi-roots'. this      *)
(* procedure searches through the positive root list,     *)
(* determining which roots are xis; if a positive root is *)
(* a xi, the 'orbitsize' field of its record is set equal *)
(* to the size of its orbit, otherwise the 'orbitsize'    *)
(* field is set to zero.                                  *)

var

root1: rootlink;
i: rankrange;
xiroot, index: boolean;
subsubweight: vector;

begin (* findxiroots *)

root1 := rootlist;
while root1 <> nil do
  begin

  (* determine if 'root1^' is a xi-root. it is a xi-root *)
  (* iff subweight [i] = 0 implies root1^. omega [i] >= 0 *)
  (* for all i.                                           *)

  xiroot := true;
  for i := 1 to rank do
    xiroot := xiroot and
      ((subweight [i] <> 0) or (root1^. omega [i] >= 0));
  if not xiroot
    then root1^. orbitsize := 0
    else

      (* 'root1^' is a xi-root; determine its orbit size. *)

      begin
      for i := 1 to rank do
        if (subweight [i] = 0) and (root1^. omega [i] = 0)
          then subsubweight [i] := 0
          else subsubweight [i] := 1;
      index := true;
      for i := 1 to rank do
```

```
        index := index and
          ((subweight [i] = 0) or (root1^. alpha[i] = 0));
      if index
        then root1^. orbitsize := subgrouporder div
             stabilizerorder (subsubweight)
        else root1^. orbitsize := 2 * subgrouporder div
             stabilizerorder (subsubweight);
      end;

    root1 := root1^. nextroot;
    end;

end; (* findxiroots *)
```

```
procedure reflectdominant (factor1: vector; var dominant1:
  vector);

(* given an arbitrary weight 'factor1', this procedure    *)
(* computes 'dominant1', the unique dominant weight       *)
(* conjugate to 'factor1' under the weyl group; since the *)
(* two weights are conjugate, 'dominant1' has the same    *)
(* multiplicity as 'factor1'.  'dominant1' is determined  *)
(* by applying successive reflections to 'factor1' until  *)
(* the result is dominant: if the coefficient of some     *)
(* fundamental weight is negative, we apply the           *)
(* reflection defined by the corresponding simple root.   *)
(* this process continues until we obtain a dominant      *)
(* weight.                                                *)

var

dominant: boolean;
i, j: rankrange;
coefficient: integer;

begin (* reflectdominant *)

dominant1 := factor1;
repeat
  dominant := true;
  for i := 1 to rank do
    if dominant1 [i] < 0
      then
        (* the coefficient of the fundamental weight is   *)
        (* negative.                                      *)
        begin
        dominant := false;

        (* apply the reflection defined by the            *)
        (* corresponding simple root; i.e. subtract       *)
        (* 'coefficient' times the simple root.           *)

        coefficient := dominant1 [i];
        for j := 1 to rank do
          dominant1 [j] := dominant1 [j] -
            coefficient * cartanmatrix [i, j];
        end;
until dominant;

end; (* reflectdominant *)
```

```
function previousmultiplicity (dominant1: vector):
  nonnegative;

(* this function first searches in the list of dominant   *)
(* weights for the weight 'dominant1'.  if 'dominant1' is *)
(* found, the function returns its multiplicity (which    *)
(* will have been previously computed); otherwise the     *)
(* function returns 0 (since if 'dominant1' is not in the *)
(* list its multiplicity is 0).                           *)

var

found: boolean;
weight1: weightlink;

begin (* previousmultiplicity *)

weight1 := weightlist;
found := false;
while (weight1 <> nil) and not found do
  begin
  found := equalvectors (weight1^. omega, dominant1);
  if not found
    then weight1 := weight1^. nextweight;
  end;
if found
  then previousmultiplicity := weight1^. multiplicity
  else previousmultiplicity := 0;

end; (* previousmultiplicity *)
```

```
function doublesummation (subweight: vector): nonnegative;

(* this function computes and returns the value of the      *)
(* double summation in the modified freudenthal formula.    *)
(* the outer sum is over the orbits of the root system;     *)
(* this sum is implemented by searching through the root    *)
(* list for the xi-roots (the orbit representatives). if    *)
(* a xi-root is encountered, the inner sum is computed;     *)
(* it is a finite sum of terms of the form scalar product   *)
(* times multiplicity. as soon as a zero term is            *)
(* encountered in the inner sum, we know that all further   *)
(* terms are zero. instead of using the actual              *)
(* multiplicity of 'factor1', which may not be available    *)
(* (since 'factor1' may not be a dominant weight), we       *)
(* reflect 'factor1' into the dominant chamber using        *)
(* procedure 'reflectdominant', and then call function      *)
(* 'previousmultiplicity' to find the multiplicity of the   *)
(* weight 'dominant1' computed by 'reflectdominant'.        *)
(* note that since 'dominant1' is a dominant weight of      *)
(* higher level than 'subweight', the multiplicity of       *)
(* 'dominant1' will have been computed by the time we       *)
(* call 'previousmultiplicity'.                             *)

var

root1: rootlink;
innerterm, innersum, result: integer;
count: nonnegative;
i: rankrange;
factor1, dominant1: vector;

begin (* doublesummation *)

result := 0;
root1 := rootlist;
while root1 <> nil do
  begin
  if root1^. orbitsize <> 0
    then
      begin
      count := 0;
      innersum := 0;
      repeat
        count := count + 1;
        for i := 1 to rank do
          factor1 [i] :=
            subweight [i] + count * root1^. omega [i];
        reflectdominant (factor1, dominant1);
        innerterm := previousmultiplicity (dominant1);
        if innerterm <> 0
          then innerterm := innerterm *
                 scalarproduct (factor1, root1^. alpha);
        innersum := innersum + innerterm;
      until innerterm = 0;
```

```
        result := result + root1^. orbitsize * innersum;
        end;
    root1 := root1^. nextroot;
    end;
doublesummation := result;

end; (* doublesummation *)
```

```
procedure computemultiplicities;

(* this procedure computes and outputs the multiplicities *)
(* of the dominant weights in the weight list.  we know   *)
(* that the multiplicity of the highest weight is one,    *)
(* and from this the multiplicities of weights with lower *)
(* levels are computed recursively using the modified     *)
(* freudenthal formula.  note that in order to obtain the *)
(* correct multiplicity for a given weight, we must       *)
(* divide the value computed by function                  *)
(* 'doublesummation' by the difference of 'highproduct'   *)
(* and 'subproduct'. the total dimension of the           *)
(* representation is computed (based on the               *)
(* multiplicities of the dominant weights and the sizes   *)
(* of their orbits) and stored in 'dimension'.  this is   *)
(* done so that the computations can be checked by        *)
(* comparing the result obtained by this method with the  *)
(* result given by the weyl dimension formula.            *)

var

weylgrouporder, subgrouporder, dimension, highproduct,
  subproduct: nonnegative;
i: rankrange;
weight1: weightlink;
omega, alpha: vector;
count: nonnegative;

begin (* computemultiplicities *)

(* determine 'weylgrouporder', the order of the weyl      *)
(* group of the algebra.                                  *)

case latype of
  'a':  weylgrouporder := order [a, rank];
  'b', 'c': weylgrouporder := order [bc, rank];
  'd':  weylgrouporder := order [d, rank];
  'e':  case rank of
          6: weylgrouporder := e6order;
          7: weylgrouporder := e7order;
          8: weylgrouporder := e8order;
        end;
  'f':  weylgrouporder := f4order;
  'g':  weylgrouporder := g2order;
  end;

(* compute 'highproduct'                                  *)

for i := 1 to rank do
  omega [i] := weightlist^. omega [i] + 1;
basischange (omega, alpha);
highproduct := scalarproduct (omega, alpha);

(* output the headings, the highest weight and its        *)
```

```
(* multiplicity, and initialize 'dimension'.                    *)

write ('       level');
for i := 1 to rank - 1 do
  write ('    ');
writeln ('weight   multiplicity');
writeln;
weightlist^. multiplicity := 1;
count := 1;
write (count: 3, '    ', weightlist^. level: 6, '    ');
for i := 1 to rank do
  write (weightlist^. omega [i]: 3);
writeln ('    ', weightlist^. multiplicity: 12);
dimension :=
  weylgrouporder div stabilizerorder (weightlist^. omega);

(* compute and output the other multiplicities.                 *)

weight1 := weightlist^. nextweight;
while weight1 <> nil do
  begin

  (* compute 'subproduct'.                                      *)

  for i := 1 to rank do
    omega [i] := weight1^. omega [i] + 1;
  basischange (omega, alpha);
  subproduct := scalarproduct (omega, alpha);

  (* compute and output the multiplicity, and increment         *)
  (* 'dimension'.                                               *)

  subgrouporder := stabilizerorder (weight1^. omega);
  findxiroots (weight1^. omega, subgrouporder);
  weight1^. multiplicity :=
    doublesummation (weight1^. omega) div
    (highproduct - subproduct);
  count := count + 1;
  write (count: 3, '    ', weight1^. level: 6, '    ');
  for i := 1 to rank do
    write (weight1^. omega [i]: 3);
  writeln ('    ', weight1^. multiplicity: 12);
  dimension := dimension + weight1^. multiplicity *
    (weylgrouporder div subgrouporder);

  weight1 := weight1^.nextweight;
  end;
writeln;

(* output the total dimension.                                  *)

writeln ('dimension of representation:');
writeln;
writeln (dimension: 1);
```

```
writeln;
end; (* computemultipicities *)
```

```
procedure weyldimensionformula (weight: vector);

(* this procedure computes the total dimension of the      *)
(* representation with highest weight 'weight', using the  *)
(* weyl dimension formula.  in order to allow the          *)
(* computations to remain integral, the running product    *)
(* is stored in terms of its prime factorization in the    *)
(* array 'power'.                                          *)

var

n: 1..numprimes;
numerator, denominator: integer;
count, result: nonnegative;
i: rankrange;
power: array [1..numprimes] of integer;
root1: rootlink;

begin (* weyldimensionformula *)

(* initialize the array of powers.                         *)

for n := 1 to numprimes do
  power [n] := 0;

(* compute the product of the factors, one factor for      *)
(* each positive root.                                     *)

root1 := rootlist;
while root1 <> nil do
  begin

  (* compute the numerator and denominator of this        *)
  (* factor.                                              *)

  numerator := 0;
  for i := 1 to rank do
    numerator := numerator +
      (weight [i] + 1) * root1^. alpha [i] *
      squarelength [i];
  numerator := numerator div 2;
  denominator := 0;
  for i := 1 to rank do
    denominator := denominator +
      root1^. alpha [i] * squarelength [i];
  denominator := denominator div 2;

  (* multiply by the numerator and divide by the          *)
  (* denominator.                                         *)

  n := 1;
  while numerator <> 1 do
    if numerator mod prime [n] <> 0
      then n := n + 1
```

```
        else
          begin
          power [n] := power [n] + 1;
          numerator := numerator div prime [n];
          end;
    n := 1;
    while denominator <> 1 do
      if denominator mod prime [n] <> 0
        then n := n+ 1
        else
          begin
          power [n] := power [n] - 1;
          denominator := denominator div prime [n];
          end;

    root1 := root1^. nextroot;
    end;

(* compute the result                                    *)

result := 1;
for n := 1 to numprimes do
  for count := 1 to power [n] do
    result := result * prime [n];
writeln ('weyl dimension formula gives');
writeln;
writeln (result: 1);
writeln;

end; (* weyldimensionformula *)
```

```
begin (* onerepresentation: main block *)

writeln;
readtyperank;
while (latype <> 'a') or (rank <> 0) do
  begin
  displaydiagram;
  matrixinitialize;
  lengthinitialize;
  orderinitialize;
  rootlist := nil;
  writeln ('compute the positive roots ?');
  if positiveanswer
    then
      begin
      computepositiveroots;
      writeln ('number of positive roots: ', numroots: 1);
      writeln ('display the positive roots ?');
      if positiveanswer
        then displaypositiveroots;
      writeln ('compute the dominant weights ?');
      if positiveanswer
        then
          begin
          weightlist := nil;
          repeat
            computedominantweights;
            sortdominantweights;
            writeln ('number of dominant weights: ',
              numweights: 1);
            writeln ('display the dominant weights ?');
            if positiveanswer
              then displaydominantweights;
            writeln ('compute the multiplicities ?');
            if positiveanswer
              then computemultiplicities;
            weyldimensionformula (weightlist^. omega);
            while weightlist <> nil do
              begin
              tempweight := weightlist^. nextweight;
              dispose (weightlist);
              weightlist := tempweight;
              end;
            writeln
              ('continue with ', latype, rank: 1, ' ?');
          until not positiveanswer;
        end;
      end;
  while rootlist <> nil do
    begin
    temproot := rootlist^. nextroot;
    dispose (rootlist);
    rootlist := temproot;
    end;
```

```
    readtyperank;
    end;

end. (* onerepresentation: main block *)
```

Examples of output

This section contains examples of the interactive execution of program ONEREPRESENTATION. The dominant weights and their multiplicities are computed for one representation each of the algebras E6, E7, E8, F4, and G2. (The program can of course also be used for algebras of types A, B, C, and D.) The timing of each computation (to the nearest hundredth of a second) appears at the end of the listing for that execution of the program. (The examples were originally generated in upper-case and have been left in that form.)

```
INPUT THE NAME OF THE LIE ALGEBRA.
A2-8, B2-8, C2-8, D4-8, E6-8, F4, G2, A0 TO STOP:
? E6

THE DYNKIN DIAGRAM (BOURBAKI NUMBERING)

          2
  1  3  4  5  6

COMPUTE THE POSITIVE ROOTS ?
Y/N:
? Y

NUMBER OF POSITIVE ROOTS: 36
DISPLAY THE POSITIVE ROOTS ?
Y/N:
? N

COMPUTE THE DOMINANT WEIGHTS ?
Y/N:
? Y

INPUT THE HIGHEST WEIGHT IN THE BASIS OF FUNDAMENTAL
DOMINANT WEIGHTS (BOURBAKI NUMBERING) ON ONE LINE.
? 2 2 0 0 0 2

IS IT CORRECT ?
Y/N
? Y

NUMBER OF DOMINANT WEIGHTS: 65
DISPLAY THE DOMINANT WEIGHTS ?
Y/N:
? N

COMPUTE THE MULTIPLICITIES ?
Y/N:
? Y
```

	LEVEL	WEIGHT						MULTIPLICITY
1	109	2	2	0	0	0	2	1
2	107	2	2	0	0	1	0	1
3	107	2	0	0	1	0	2	1
4	107	0	2	1	0	0	2	1
5	105	2	0	0	1	1	0	1
6	105	0	2	1	0	1	0	1
7	105	0	0	1	1	0	2	1
8	103	0	0	1	1	1	0	1
9	101	2	1	1	0	0	1	4
10	101	1	1	0	0	1	2	4
11	99	1	3	0	0	0	1	5
12	99	0	1	2	0	0	1	4
13	99	1	1	0	0	2	0	4

14	97	1	1	0	1	0	1	14
15	97	3	0	0	0	0	3	3
16	95	3	0	0	0	1	1	9
17	95	1	0	1	0	0	3	9
18	93	1	0	1	0	1	1	27
19	93	3	2	0	0	0	0	10
20	93	2	0	2	0	0	0	10
21	93	0	2	0	0	0	3	10
22	93	0	0	0	0	2	2	10
23	91	1	2	1	0	0	0	32
24	91	0	2	0	0	1	1	32
25	91	3	0	0	1	0	0	19
26	91	0	0	0	1	0	3	19
27	91	0	0	3	0	0	0	10
28	91	0	0	0	0	3	0	10
29	89	1	0	1	1	0	0	56
30	89	0	0	0	1	1	1	56
31	89	0	4	0	0	0	0	15
32	87	2	1	0	0	0	2	62
33	87	0	2	0	1	0	0	69
34	85	2	1	0	0	1	0	113
35	85	0	1	1	0	0	2	113
36	85	0	0	0	2	0	0	114
37	83	0	1	1	0	1	0	206
38	81	4	0	0	0	0	1	36
39	81	1	0	0	0	0	4	36
40	79	2	0	1	0	0	1	212
41	79	1	0	0	0	1	2	212
42	77	1	2	0	0	0	1	385
43	77	0	0	2	0	0	1	362
44	77	1	0	0	0	2	0	362
45	75	1	0	0	1	0	1	652
46	71	3	1	0	0	0	0	360
47	71	0	1	0	0	0	3	360
48	69	1	1	1	0	0	0	1120
49	69	0	1	0	0	1	1	1120
50	67	0	3	0	0	0	0	705
51	65	2	0	0	0	0	2	1140
52	65	0	1	0	1	0	0	1911
53	63	2	0	0	0	1	0	1890
54	63	0	0	1	0	0	2	1890
55	61	0	0	1	0	1	0	3123
56	55	1	1	0	0	0	1	5113
57	49	3	0	0	0	0	0	3120
58	49	0	0	0	0	0	3	3120
59	47	1	0	1	0	0	0	8200
60	47	0	0	0	0	1	1	8200
61	45	0	2	0	0	0	0	6300
62	43	0	0	0	1	0	0	13061
63	33	1	0	0	0	0	1	20491
64	23	0	1	0	0	0	0	31915
65	1	0	0	0	0	0	0	49320

DIMENSION OF REPRESENTATION:

```
66023100

WEYL DIMENSION FORMULA GIVES

66023100

CONTINUE WITH E6 ?
Y/N:
? N

INPUT THE NAME OF THE LIE ALGEBRA.
A2-8, B2-8, C2-8, D4-8, E6-8, F4, G2, A0 TO STOP:
? A0

TIME:   9.76 SECONDS
```

```
INPUT THE NAME OF THE LIE ALGEBRA.
A2-6, B2-8, C2-6, D4-6, E6-8, F4, G2, A0 TO STOP:
? E7

THE DYNKIN DIAGRAM (BOURBAKI NUMBERING)

        2
  1  3  4  5  6  7

COMPUTE THE POSITIVE ROOTS ?
Y/N:
? Y

NUMBER OF POSITIVE ROOTS: 63
DISPLAY THE POSITIVE ROOTS ?
Y/N:
? N

COMPUTE THE DOMINANT WEIGHTS ?
Y/N:
? Y

INPUT THE HIGHEST WEIGHT IN THE BASIS OF FUNDAMENTAL
DOMINANT WEIGHTS (BOURBAKI NUMBERING) ON ONE LINE.
? 0 0 1 0 1 1 0

IS IT CORRECT ?
Y/N:
? Y

NUMBER OF DOMINANT WEIGHTS: 64
DISPLAY THE DOMINANT WEICHTS ?
Y/N:
? N

COMPUTE THE MULTIPLICITIES ?
Y/N:
? Y
```

LEVEL		WEIGHT							MULTIPLICITY
1	194	0	0	1	0	1	1	0	1
2	190	0	0	1	1	0	0	1	2
3	188	1	1	0	0	0	2	0	3
4	186	1	1	0	0	1	0	1	6
5	182	0	1	2	0	0	0	0	5
6	180	1	0	1	0	0	1	1	16
7	180	0	0	0	0	1	2	0	10
8	180	1	1	0	1	0	0	0	13
9	178	0	0	0	0	2	0	1	20
10	178	0	2	0	0	0	1	1	15
11	176	1	0	1	0	1	0	0	36
12	176	0	0	0	1	0	1	1	44
13	174	0	2	0	0	1	0	0	33

14	172	0	0	0	1	1	0	0	96
15	172	2	1	0	0	0	0	2	36
16	170	2	1	0	0	0	1	0	61
17	170	0	1	1	0	0	0	2	92
18	168	0	1	1	0	0	1	0	207
19	166	1	0	0	0	0	2	1	104
20	164	1	0	0	0	1	0	2	204
21	162	1	0	0	0	1	1	0	460
22	162	2	0	1	0	0	0	1	175
23	160	1	2	0	0	0	0	1	426
24	160	0	0	2	0	0	0	1	435
25	158	1	0	0	1	0	0	1	911
26	156	0	1	0	0	0	1	2	425
27	154	0	1	0	0	0	2	0	970
28	152	0	1	0	0	1	0	1	1857
29	152	3	1	0	0	0	0	0	360
30	150	1	1	1	0	0	0	0	1785
31	150	2	0	0	0	0	0	3	416
32	148	2	0	0	0	0	1	1	1817
33	148	0	3	0	0	0	0	0	861
34	146	0	0	1	0	0	0	3	836
35	146	0	0	1	0	0	1	1	3620
36	146	0	1	0	1	0	0	0	3555
37	144	2	0	0	0	1	0	0	3480
38	142	0	0	1	0	1	0	0	6816
39	138	1	1	0	0	0	0	2	6807
40	136	1	1	0	0	0	1	0	12692
41	134	0	0	0	0	0	1	3	1600
42	132	0	0	0	0	0	2	1	6900
43	130	3	0	0	0	0	0	1	6570
44	130	0	0	0	0	1	0	2	12600
45	128	0	0	0	0	1	1	0	23280
46	128	1	0	1	0	0	0	1	23100
47	126	0	2	0	0	0	0	1	23157
48	124	0	0	0	1	0	0	1	41571
49	118	2	1	0	0	0	0	0	41325
50	116	0	1	1	0	0	0	0	73255
51	116	1	0	0	0	0	0	3	22702
52	114	1	0	0	0	0	1	1	73211
53	110	1	0	0	0	1	0	0	127251
54	104	0	1	0	0	0	0	2	126828
55	102	0	1	0	0	0	1	0	217948
56	96	2	0	0	0	0	0	1	217840
57	94	0	0	1	0	0	0	1	368305
58	84	1	1	0	0	0	0	0	615265
59	82	0	0	0	0	0	0	3	216378
60	80	0	0	0	0	0	1	1	614624
61	76	0	0	0	0	1	0	0	1015785
62	62	1	0	0	0	0	0	1	1659430
63	50	0	1	0	0	0	0	0	2684430
64	28	0	0	0	0	0	0	1	4301640

DIMENSION OF REPRESENTATION:

```
29302560000

WEYL DIMENSION FORMULA GIVES

29302560000

CONTINUE WITH E7 ?
Y/N:
? N

INPUT THE NAME OF THE LIE ALGEBRA.
A2-8, B2-8, C2-8, D4-8, E6-8, F4, G2, A0 TO STOP:
? A0

TIME:  12.76 SECONDS
```

```
INPUT THE NAME OF THE LIE ALGEBRA.
A2-8, B2-8, C2-8, D4-8, E6-8, F4, G2, A0 TO STOP:
? E6

THE DYNKIN DIAGRAM (BOURBAKI NUMBERING)

          2
  1   3   4   5   6   7   8

COMPUTE THE POSITIVE ROOTS ?
Y/N:
? Y

NUMBER OF POSITIVE ROOTS: 120
DISPLAY THE POSITIVE ROOTS ?
Y/N:
? N

COMPUTE THE DOMINANT WEIGHTS ?
Y/N:
? Y

INPUT THE HIGHEST WEIGHT IN THE BASIS OF FUNDAMENTAL
DOMINANT WEIGHTS (BOURBAKI NUMBERING) ON ONE LINE.
? 1 1 0 0 0 0 0 2

IS IT CORRECT ?
Y/N:
? Y

NUMBER OF DOMINANT WEIGHTS: 58
DISPLAY THE DOMINANT WEIGHTS ?
Y/N:
? N

COMPUTE THE MULTIPLICITIES ?
Y/N:
? Y
```

LEVEL		WEIGHT								MULTIPLICITY
1	345	1	1	0	0	0	0	0	2	1
2	343	1	1	0	0	0	0	1	0	1
3	337	0	0	0	0	1	0	0	2	4
4	335	0	0	0	0	1	0	1	0	4
5	333	1	0	1	0	0	0	0	1	6
6	331	0	2	0	0	0	0	0	1	7
7	329	0	0	0	1	0	0	0	1	22
8	323	1	0	0	0	0	0	1	2	16
9	321	1	0	0	0	0	0	2	0	16
10	321	2	1	0	0	0	0	0	0	21
11	319	1	0	0	0	0	1	0	1	72
12	319	0	1	1	0	0	0	0	0	72
13	313	1	0	0	0	1	0	0	0	204

14	311	0	1	0	0	0	0	0	3	56
15	309	0	1	0	0	0	0	1	1	216
16	305	0	1	0	0	0	1	0	0	555
17	301	2	0	0	0	0	0	0	2	224
18	299	0	0	1	0	0	0	0	2	620
19	299	2	0	0	0	0	0	1	0	544
20	297	0	0	1	0	0	0	1	0	1410
21	289	0	0	0	0	0	0	1	3	171
22	287	1	1	0	0	0	0	0	1	3405
23	287	0	0	0	0	0	0	2	1	603
24	285	0	0	0	0	0	1	0	2	1622
25	283	0	0	0	0	0	1	1	0	3430
26	279	0	0	0	0	1	0	0	1	7932
27	277	3	0	0	0	0	0	0	0	1344
28	275	1	0	1	0	0	0	0	0	7770
29	273	0	2	0	0	0	0	0	0	7640
30	271	0	0	0	1	0	0	0	0	17440
31	267	1	0	0	0	0	0	0	3	4036
32	265	1	0	0	0	0	0	1	1	17803
33	261	1	0	0	0	0	1	0	0	37650
34	253	0	1	0	0	0	0	0	2	36724
35	251	0	1	0	0	0	0	1	0	79118
36	243	2	0	0	0	0	0	0	1	78744
37	241	0	0	1	0	0	0	0	1	161688
38	233	0	0	0	0	0	0	0	4	9576
39	231	0	0	0	0	0	0	1	2	81945
40	229	1	1	0	0	0	0	0	0	322252
41	229	0	0	0	0	0	0	2	0	162360
42	227	0	0	0	0	0	1	0	1	323946
43	221	0	0	0	0	1	0	0	0	631564
44	209	1	0	0	0	0	0	0	2	635616
45	207	1	0	0	0	0	0	1	0	1213994
46	195	0	1	0	0	0	0	0	1	2293676
47	185	2	0	0	0	0	0	0	0	2291408
48	183	0	0	1	0	0	0	0	0	4262734
49	175	0	0	0	0	0	0	0	3	1224720
50	173	0	0	0	0	0	0	1	1	4267616
51	169	0	0	0	0	0	1	0	0	7813194
52	151	1	0	0	0	0	0	0	1	14126812
53	137	0	1	0	0	0	0	0	0	25220384
54	117	0	0	0	0	0	0	0	2	25226152
55	115	0	0	0	0	0	0	1	0	44502550
56	93	1	0	0	0	0	0	0	0	77061920
57	59	0	0	0	0	0	0	0	1	134129408
58	1	0	0	0	0	0	0	0	0	229393920

DIMENSION OF REPRESENTATION:

3191795712000

WEYL DIMENSION FORMULA GIVES

3191795712000

```
CONTINUE WITH E8 ?
Y/N:
? N

INPUT THE NAME OF THE LIE ALGEBRA.
A2-8, B2-8, C2-8, D4-8, E6-8, F4, G2, A0 TO STOP:
? A0

TIME:  12.53 SECONDS
```

```
INPUT THE NAME OF THE LIE ALGEBRA.
A2-8, B2-8, C2-8, D4-8, E6-8, F4, G2, A0 TO STOP:
? F4

THE DYNKIN DIAGRAM (BOURBAKI NUMBERING)

  1  2  >  3  4

COMPUTE THE POSITIVE ROOTS ?
Y/N:
? Y

NUMBER OF POSITIVE ROOTS: 24
DISPLAY THE POSITIVE ROOTS ?
Y/N:
? N

COMPUTE THE DOMINANT WEIGHTS ?
Y/N:
? Y

INPUT THE HIGHEST WEIGHT IN THE BASIS OF FUNDAMENTAL
DOMINANT WEIGHTS (BOURBAKI NUMBERING) ON ONE LINE.
? 1 1 1 1

IS IT CORRECT ?
Y/N:
? Y

NUMBER OF DOMINANT WEIGHTS: 58
DISPLAY THE DOMINANT WEIGHTS ?
Y/N:
? N

COMPUTE THE MULTIPLICITIES ?
Y/N:
? Y
```

	LEVEL	WEIGHT				MULTIPLICITY
1	111	1	1	1	1	1
2	107	0	0	3	1	2
3	107	2	0	1	2	2
4	107	1	2	0	0	2
5	105	0	1	1	2	4
6	105	2	0	2	0	4
7	103	2	1	0	1	6
8	103	0	1	2	0	6
9	101	1	0	1	3	6
10	101	0	2	0	1	12
11	99	1	0	2	1	22
12	99	3	0	0	2	8
13	97	1	1	0	2	30
14	97	3	0	1	0	12

15	95	1	1	1	0	44
16	95	0	0	1	4	14
17	93	0	0	2	2	52
18	93	2	0	0	3	40
19	91	2	0	1	1	80
20	91	0	0	3	0	76
21	91	0	1	0	3	68
22	89	0	1	1	1	136
23	89	4	0	0	0	16
24	87	2	1	0	0	108
25	87	1	0	0	4	88
26	85	1	0	1	2	228
27	85	0	2	0	0	164
28	83	1	0	2	0	308
29	83	3	0	0	1	144
30	81	1	1	0	1	400
31	81	0	0	0	5	112
32	79	0	0	1	3	370
33	77	0	0	2	1	644
34	77	2	0	0	2	512
35	75	0	1	0	2	816
36	75	2	0	1	0	664
37	73	0	1	1	0	1052
38	71	1	0	0	3	1024
39	69	1	0	1	1	1664
40	67	3	0	0	0	848
41	65	1	1	0	0	2094
42	65	0	0	0	4	1280
43	63	0	0	1	2	2572
44	61	2	0	0	1	2616
45	61	0	0	2	0	3224
46	59	0	1	0	1	3992
47	55	1	0	0	2	4920
48	53	1	0	1	0	6072
49	49	0	0	0	3	6032
50	47	0	0	1	1	9086
51	45	2	0	0	0	7456
52	43	0	1	0	0	11088
53	39	1	0	0	1	13480
54	33	0	0	0	2	16336
55	31	0	0	1	0	19764
56	23	1	0	0	0	23840
57	17	0	0	0	1	28688
58	1	0	0	0	0	34432

DIMENSION OF REPRESENTATION:

16777216

WEYL DIMENSION FORMULA GIVES

16777216

CONTINUE WITH F4 ?

```
Y/N:
? N

INPUT THE NAME OF THE LIE ALGEBRA.
A2-8, B2-8, C2-6, D4-8, E6-8, F4, G2, A0 TO STOP:
? A0

TIME:   9.38 SECONDS
```

```
INPUT THE NAME OF THE LIE ALGEBRA.
A2-8, B2-8, C2-8, D4-8, E6-8, F4, G2, A0 TO STOP:
? G2

THE DYNKIN DIAGRAM (BOURBAKI NUMBERING)

  1 < 2

COMPUTE THE POSITIVE ROOTS ?
Y/N:
? Y

NUMBER OF POSITIVE ROOTS: 6
DISPLAY THE POSITIVE ROOTS ?
Y/N:
? N

COMPUTE THE DOMINANT WEIGHTS ?
Y/N:
? Y

INPUT THE HIGHEST WEIGHT IN THE BASIS OF FUNDAMENTAL
DOMINANT WEIGHTS (BOURBAKI NUMBERING) ON ONE LINE.
? 5 5

IS IT CORRECT ?
Y/N:
? Y

NUMBER OF DOMINANT WEIGHTS: 63
DISPLAY THE DOMINANT WEIGHTS ?
Y/N:
? N

COMPUTE THE MULTIPLICITIES ?
Y/N:
? Y
```

	LEVEL	WEIGHT		MULTIPLICITY
1	81	5	5	1
2	79	8	3	1
3	79	3	6	1
4	77	6	4	2
5	77	11	1	1
6	77	1	7	1
7	75	4	5	3
8	75	9	2	2
9	73	2	6	4
10	73	7	3	4
11	73	12	0	2
12	71	5	4	7
13	71	0	7	4
14	71	10	1	4

15	69	8	2	8
16	69	3	5	9
17	67	6	3	12
18	67	1	6	10
19	67	11	0	8
20	65	4	4	16
21	65	9	1	13
22	63	2	5	19
23	63	7	2	19
24	61	5	3	26
25	61	0	6	20
26	61	10	0	20
27	59	8	1	29
28	59	3	4	31
29	57	6	2	38
30	57	1	5	34
31	55	4	3	46
32	55	9	0	40
33	53	2	4	52
34	53	7	1	52
35	51	5	2	64
36	51	0	5	54
37	49	8	0	68
38	49	3	3	73
39	47	6	1	83
40	47	1	4	78
41	45	4	2	96
42	43	2	3	105
43	43	7	0	102
44	41	5	1	120
45	41	0	4	108
46	39	3	2	133
47	37	6	0	142
48	37	1	3	140
49	35	4	1	160
50	33	2	2	172
51	31	5	0	184
52	31	0	3	176
53	29	3	1	201
54	27	1	2	210
55	25	4	0	224
56	23	2	1	239
57	21	0	2	244
58	19	3	0	260
59	17	1	1	271
60	13	2	0	286
61	11	0	1	294
62	7	1	0	306
63	1	0	0	312

DIMENSION OF REPRESENTATION:

46656

WEYL DIMENSION FORMULA GIVES

46656

CONTINUE WITH G2 ?
Y/N:
? N

INPUT THE NAME OF THE LIE ALGEBRA.
A2-8, B2-6, C2-6, D4-8, E6-6, F4, G2, A0 TO STOP:
? A0

TIME: 6.84 SECONDS

Chapter 5.

MULTIPLICITYTABLE:
A non-interactive program which computes the multiplicities for many lower-dimensional representations of a given Lie algebra

Introduction

The non-interactive program MULTIPLICITYTABLE computes the dominant weights and their multiplicities for many lower-dimensional representations of a given Lie algebra, and outputs this information in a table together with the order of the stabilizer of each highest weight, the size of the orbit of each highest weight under the Weyl group, the total dimension of the representation associated with each highest weight, and the square length and level of each highest weight. This chapter presents an overview of the program, followed by a detailed discussion of its data structures and procedures, a listing of the program, and some examples of its output.

The finite-dimensional irreducible representations of a simple Lie algebra divide naturally into congruence classes with the property that all the weights in the weight system of a given representation lie in the same class as the highest weight of the representation. Thus it is

appropriate to consider only one congruence class for each table. The number of congruence classes equals the determinant of the Cartan matrix, and they are conventionally numbered beginning with 0 (see Lemire and Patera 1980).

Execution begins with the reading of the input data: the type and rank of the Lie algebra, and the congruence class and number of representations (equivalently, highest weights) to be included in the table; for example, algebra E6, class 0, 50 representations. Initializations are done by MATRIXINITIALIZE, LENGTHINITIALIZE, and ORDERINITIALIZE. The positive roots of the algebra are determined by COMPUTEPOSITIVEROOTS (which calls INSERTROOTS); they are stored in a linked list with head pointer ROOTLIST. The highest weights are generated (by increasing level) by COMPUTEHIGHESTWEIGHTS, which calls COMPUTELEVELLIST; the weights are stored in a linked list with head pointer WEIGHTLIST. Computation of highest weights with greater levels continues until the required number has been generated. COMPUTEHIGHESTWEIGHTS then transfers the weights to WEIGHTARRAY and disposes of the dynamic storage used during the computation of the weights. The contents of the table are computed by COMPUTETABLE which calls FLAGSUBWEIGHTS and COMPUTEMULTIPLICITIES and (through them) various other procedures; the multiplicities are stored in the one-dimensional array MULTIARRAY. Finally the results are output by PRINTTABLE, which calls PRINTSEGMENT and

PRINTLINE.

The tables computed by program MULTIPLICITYTABLE are upper-triangular matrices of multiplicities, together with rows of additional information along the top. The program will automatically split the table into vertical segments if it is too wide to fit on one page. The remarks in paragraphs 3 and 4 of the Introduction to Chapter 4 also apply to this program. In addition, the non-standard Pascal function CLOCK is called twice by the main block.

MULTIPLICITYTABLE was used to compute the tables included in Bremner, Moody, Patera (to appear).

Documentation of the program

Files

INPUT, OUTPUT

The program is not interactive; INPUT must correspond to a file containing (in order) the type LATYPE and rank RANK of the algebra, as well as the congruence class CLASS and the number NUMREPRESENTATIONS of the representations to be included in the table. LATYPE (a character variable) must appear as the first character in the file; RANK, CLASS, and NUMREPRESENTATIONS are integers and no special format is required (except of course for separating blanks). After execution is complete, the file corresponding to OUTPUT will contain the table.

Global constants

MAXRANK

As in ONEREPRESENTATION.

MAXREPRESENTATIONS

The maximum number of representations which may be included in the table. In order to compute a larger table, the user should change MAXREPRESENTATIONS and recompile the program.

MAXMULTIPLICITIES

The maximum number of multiplicities in the table. Since for each representation i (1 ≤ i ≤ MAXREPRESENTATIONS) the table includes a column of i multiplicities (some of which may be zero), MAXMULTIPLICITIES must equal MAXREPRESENTATIONS x (MAXREPRESENTATIONS + 1) / 2.

MAXCOLUMNS

The maximum number of columns of multiplicities in each output segment of the table. Each column is four characters wide, and the labels for the rows (along the right margin of the table) are eight characters wide. Thus, for example, if the tables are to be printed with labels on a 132-column printer, MAXCOLUMNS should be at most (132 - 8) / 4 = 31.

E6ORDER, E7ORDER, E8ORDER, F4ORDER, G2ORDER
NUMPRIMES

As in ONEREPRESENTATION.

Global types

```
NONNEGATIVE = 0..MAXINT;
CLASSICALTYPES = (A, BC, D);
RANKRANGE = 1..MAXRANK;
```

As in ONEREPRESENTATION.

```
REPRESENTATIONRANGE = 1..MAXREPRESENTATIONS
MULTIRANGE = 1..MAXMULTIPLICITIES
```

Subranges for array indices.

```
CLASSRANGE = 0..MAXRANK
```

Subrange for congruence class numbers.

```
VECTOR = ARRAY [RANKRANGE] OF INTEGER
```

As in ONEREPRESENTATION.

```
ROWTYPE = ARRAY [REPRESENTATIONRANGE] OF NONNEGATIVE
```

This type is used to define arrays containing a row of the multiplicity table.

```
ROOTLINK = ROOT;
ROOT =
  RECORD
  ALPHA, OMEGA: VECTOR;
  LEVEL, ORBITSIZE: NONNEGATIVE;
  NEXTROOT: ROOTLINK;
  END;
WEIGHTLINK = WEIGHT;
WEIGHT =
  RECORD
  OMEGA: VECTOR;
  MULTIPLICITY, DEPTH, LEVEL: NONNEGATIVE;
  NEXTWEIGHT: WEIGHTLINK;
  END;
LINK = NODE;
NODE =
  RECORD
  LEFT, RIGHT: LINK;
  NONZERO: BOOLEAN;
  END;
```

As in ONEREPRESENTATION.

Global variables

```
CARTANMATRIX, INVERSEMATRIX: ARRAY [RANKRANGE, RANKRANGE] OF
  INTEGER;
DETERMINANT: NONNEGATIVE;
SQUARELENGTH: VECTOR;
LATYPE: CHAR;
RANK: INTEGER;
```

As in ONEREPRESENTATION.

```
CLASS: INTEGER;
NUMREPRESENTATIONS: INTEGER;
```

The congruence class and number of representations to be included in the table.

```
ROOTLIST: ROOTLINK;
```

As in ONEREPRESENTATION.

```
WEIGHTARRAY: ARRAY [REPRESENTATIONRANGE] OF WEIGHT;
```

An array which holds the list of highest weights. The highest weights are transferred to this array from the dynamic weight list after they have been generated; this is done so that during the computation by procedure FLAGSUBWEIGHTS of the dominant weights in each representation the program can immediately access each weight given its column in the table.

```
MULTIARRAY: ARRAY [MULTIRANGE] OF NONNEGATIVE;
```

An array which holds the weight multiplicities. The multiplicities are output in an upper-triangular table of

NUMREPRESENTATIONS rows (and the same number of columns). For $1 \leq i \leq$ NUMREPRESENTATIONS, row i contains NUMREPRESENTATIONS - i + 1 multiplicities. In MULTIARRAY, the table is represented as a linear array with MAXMULTIPLICITIES elements by placing each row immediately to the right of the preceding row. This arrangement was chosen in order to save space; if a rectangular array was chosen instead, addressing of entries would be simpler but the lower half of the array (below the main diagonal) would never be used. Each element of MULTIARRAY is initialized to zero by the VALUE segment.

STABARRAY, ORBITSIZEARRAY, DIMENSIONARRAY, PRODUCTARRAY, LEVELARRAY: ROWTYPE;

Arrays which hold data about the corresponding weights in WEIGHTARRAY. STABARRAY contains the order of the stabilizer of the weight. ORBITSIZEARRAY contains the size of the orbit of the weight under the Weyl group. DIMENSIONARRAY contains the dimension of the representation with the weight as highest weight. PRODUCTARRAY contains the scalar product of the weight with itself, i.e. the square-length of the weight. LEVELARRAY contains the level of the weight.

CARRIER: ARRAY [RANKRANGE] OF LINK;
ORDER: ARRAY [CLASSICALTYPES, RANKRANGE] OF NONNEGATIVE;
PRIME: ARRAY [1..NUMPRIMES] OF NONNEGATIVE;

As in ONEREPRESENTATION.

STARTTIME: INTEGER;

The time at which the program starts execution.

Procedures and functions

In the descriptions of the procedures and functions, calls to standard Pascal procedures are not mentioned with the exceptions of NEW and DISPOSE.

FUNCTION VALIDINPUT: BOOLEAN;

This function, which is called by the main block, checks the validity of the input and returns TRUE if and only if the input is valid.

```
PROCEDURE SPECIALLINEARMATRIX;
PROCEDURE MATRIXINITIALIZE;
PROCEDURE LENGTHINITIALIZE;
PROCEDURE ORDERINITIALIZE;
PROCEDURE BASISCHANGE (OMEGA: VECTOR; VAR ALPHA: VECTOR);
FUNCTION SCALARPRODUCT (VECTOR1, VECTOR2: VECTOR): INTEGER;
FUNCTION EQUALVECTORS (VECTOR1, VECTOR2: VECTOR): BOOLEAN;
PROCEDURE INSERTROOT (ROOT1: ROOTLINK; NEWROOT: VECTOR;
  NEWLEVEL: NONNEGATIVE);
PROCEDURE COMPUTEPOSITIVEROOTS;
```

These procedures and functions are identical to those with the same names in ONEREPRESENTATION. SPECIALLINEARMATRIX is called by MATRIXINITIALIZE, which is called by the main block. LENGTHINITIALIZE is called by the main block. ORDERINITIALIZE is called by the main block, and calls NEW. BASISCHANGE is called by INSERTROOT, COMPUTEPOSITIVEROOTS, COMPUTEMULTIPLICITIES, and COMPUTETABLE. SCALARPRODUCT is

called by DOUBLESUMMATION, COMPUTEMULTIPLICITIES, and COMPUTETABLE. EQUALVECTORS is called by INSERTROOT, FLAGSUBWEIGHTS, and PREVIOUSMULTIPLICITY. INSERTROOT is called by COMPUTEPOSITIVEROOTS and calls BASISCHANGE, EQUALVECTORS, and NEW. COMPUTEPOSITIVEROOTS is called by the main block and calls BASISCHANGE, INSERTROOT, and NEW.

FUNCTION CONGRUENCECLASS (WEIGHT: VECTOR): CLASSRANGE;

This function, which is called by COMPUTELEVELLIST, computes the congruence class of its argument, a weight represented relative to the omega-basis. The formulas used are from Lemire and Patera (1980).

PROCEDURE COMPUTELEVELLIST (VAR LASTWEIGHT: WEIGHTLINK; INVERSEROWSUM: VECTOR; LEVEL: NONNEGATIVE; VAR NUMWEIGHTS: INTEGER);

This procedure, which is called by COMPUTEHIGHESTWEIGHTS and includes the recursive procedure RECURSION (this is the only instance of nested procedures in the program), generates the highest weights with level LEVEL, determines which weights are in congruence class CLASS, and adds those weights to the end of the weight list (LASTWEIGHT always points the current last node in the weight list). Each time a new weight is added to the weight list, NUMWEIGHTS is incremented. The level of a weight WEIGHT is one plus twice the scalar product of INVERSEROWSUM and WEIGHT; however in this procedure the unaltered scalar product is called the LEVEL of the weight. The set of vectors WEIGHT with non-negative

integral components such that the scalar product of WEIGHT with INVERSEROWSUM is LEVEL is generated by procedure RECURSION. This procedure calls RECURSION, which calls CONGRUENCECLASS, itself, and the predefined procedure NEW.

PROCEDURE COMPUTEHIGHESTWEIGHTS;

This procedure, which is called by the main block, first initializes the vector INVERSEROWSUM which is used by COMPUTELEVELLIST to determine the levels of the weights. It then allocates a dummy head node for the weight list; WEIGHTLIST points to this head node. The highest weights are computed by increasing level by repeated calls to COMPUTELEVELLIST. The weights are then transferred from the dynamic weight list to the array WEIGHTARRAY, and finally the procedure disposes of the records in the dynamic weight list. This procedure calls COMPUTELEVELLIST and the predefined procedures NEW and DISPOSE.

```
PROCEDURE CREATECHAIN (WEIGHT: VECTOR; FIRST, LAST: LINK);
FUNCTION AORDER (VAR L1, L2: LINK): NONNEGATIVE;
FUNCTION BCORDER (VAR L1, L2: LINK): NONNEGATIVE;
FUNCTION DORDER (VAR L1, L2: LINK): NONNEGATIVE;
FUNCTION EORDER (VAR L1, L2: LINK): NONNEGATIVE;
FUNCTION FORDER (VAR L1, L2: LINK): NONNEGATIVE;
FUNCTION STABILIZERORDER (WEIGHT: VECTOR): NONNEGATIVE;
```

These procedures are identical to those with the same names in ONEREPRESENTATION. The first six are called by STABILIZERORDER; in addition, of the 'ORDER' functions, AORDER is called by the other four, BCORDER is called by FORDER, and DORDER is called by EORDER. STAPILIZERORDER is

called by FINDXIROOTS and COMPUTETABLE.

FUNCTION INDEXFUNCTION (ROW, COLUMN: REPRESENTATIONRANGE): MULTIRANGE;

This function, which is called by FLAGSUBWEIGHTS, PREVIOUSMULTIPLICITY, COMPUTEMULTIPLICITIES, and PRINTSEGMENT, computes the MULTIARRAY index corresponding to the two coordinates ROW and COLUMN of an entry in the upper-triangular table of weight multiplicities.

FUNCTION WEYLDIMENSIONFORMULA (WEIGHT: VECTOR): NONNEGATIVE;

This function, which is called by COMPUTEMULTIPLICITIES, computes the total dimension of the representation with highest weight WEIGHT, using the Weyl dimension formula. This function is very similar to the procedure of the same name in ONEREPRESENTATION; the only difference is that here WEYLDIMENSIONFORMULA is a function which returns the total dimension, whereas in ONEREPRESENTATION it is a procedure which outputs the total dimension.

PROCEDURE FLAGSUBWEIGHTS (REPNUMBER: REPRESENTATIONRANGE);

This procedure, which is called by COMPUTETABLE, computes the dominant weight system of the representation with highest weight WEIGHTARRAY [REPNUMBER]. For each dominant weight it sets the corresponding element of MULTIARRAY to one, indicating to COMPUTEMULTIPLICITIES that a multiplicity must be computed. This procedure is similar to COMPUTEDOMINANTWEIGHTS in ONEREPRESENTATION; the same

algorithm is used for computing the weights but different data structures are used in the two procedures. This procedure calls INDEXFUNCTION and EQUALVECTORS.

```
PROCEDURE FINDXIROOTS (SUBWEIGHT: VECTOR; SUBGROUPORDER:
  NONNEGATIVE);
PROCEDURE REFLECTDOMINANT (FACTOR1: VECTOR; VAR DOMINANT1:
  VECTOR);
```

These two procedures are identical to those with the same names in ONEREPRESENTATION. FINDXIROOTS is called by COMPUTEMULTIPLICITIES and calls STABILIZERORDER. REFLECTDOMINANT is called by DOUBLESUMMATION.

```
FUNCTION PREVIOUSMULTIPLICITY (DOMINANT1: VECTOR; REPNUMBER:
  REPRESENTATIONRANGE): NONNEGATIVE;
FUNCTION DOUBLESUMMATION (SUBWEIGHT: VECTOR; REPNUMBER:
  REPRESENTATIONRANGE): NONNEGATIVE;
```

These two functions are nearly identical to those with the same names in ONEREPRESENTATION; the only difference is that in this program the searching is done through the arrays WEIGHTARRAY and MULTIARRAY rather than through the linked list of weight records. PREVIOUSMULTIPLICITY is called by DOUBLESUMMATION, which is called by COMPUTEMULTIPLICITIES. PREVIOUSMULTIPLICITY calls EQUALVECTORS and INDEXFUNCTION, and DOUBLESUMMATION calls REFLECTDOMINANT, PREVIOUSMULTIPLICITY, and SCALARPRODUCT.

```
PROCEDURE COMPUTEMULTIPLICITIES (REPNUMBER:
  REPRESENTATIONRANGE);
```

This procedure, which is called by COMPUTETABLE, computes

the multiplicities of the dominant weights in the representation with highest weight WEIGHTARRAY [REPNUMBER]. This procedure is very similar to the procedure of the same name in ONEREPRESENTATION; the only differences are that here WEYLGROUPORDER is not computed (since the orbit sizes of the weights have already been determined) and the multiplicities are not output as soon as they are computed. COMPUTEMULTIPLICITIES calls INDEXFUNCTION, BASISCHANGE, SCALARPRODUCT, FINDXIROOTS, and DOUBLESUMMATION.

PROCEDURE COMPUTETABLE;

This procedure, which is called by the main block, computes the information to be included in the table: the contents of the arrays STABARRAY, ORBITSIZEARRAY, PRODUCTARRAY, DIMENSIONARRAY, and LEVELARRAY, as well as the multiplicities stored in MULTIARRAY. COMPUTETABLE calls BASISCHANGE, SCALARPRODUCT, STABILIZERORDER, FLAGSUBWEIGHTS, COMPUTEMULTIPLICITIES and WEYLDIMENSIONFORMULA.

PROCEDURE PRINTLINE (TABLEROW: ROWTYPE; LEFT, RIGHT, ROW: REPRESENTATIONRANGE; INMULTIS: BOOLEAN);

This procedure, which is called by PRINTSEGMENT, outputs the row of the table contained in TABLEROW between columns LEFT and RIGHT inclusive. If TABLEROW is a row of multiplicities, ROW is the (ordinal) number of the row; for other rows, ROW is one. INMULTIS is TRUE if and only if TABLEROW is a row of multiplicities. In order to enforce a standard column width, entries in the row are split into

'pieces' (i.e. three-digit segments), starting with the lowest three digits, and printed as a column of pieces. All of the computation done by this procedure is necessary to ensure correct formatting of the output.

```
PROCEDURE PRINTSEGMENT (LEFT, RIGHT: REPRESENTATIONRANGE;
  TITLES: BOOLEAN);
```

This procedure, which is called by PRINTTABLE, prints the segment of the table between columns LEFT and RIGHT inclusive. Labels for the rows will be printed at the right margin of the segment if and only if TITLES is true. This procedure calls INDEXFUNCTION and PRINTLINE.

```
PROCEDURE PRINTTABLE;
```

This procedure, which is called by the main block, prints the title for the table, determines how the table will be divided into (vertical) segments, and then repeatedly calls PRINTSEGMENT to print the segments.

Listing of the program

Procedures and functions in this program which are identical to those with the same names in ONEREPRESENTATION have been replaced by empty BEGIN-END blocks. Although the names of constants, types, variables, procedures and functions appear in upper-case throughout the documentation of the program, the listing of the program is entirely in lower-case for easier reading.

```
program multiplicitytable (input, output);

(* the program contains the following procedures and      *)
(* functions (in order of appearance):                    *)
(*                                                        *)
(* function  validinput                                   *)
(* procedure speciallinearmatrix                          *)
(* procedure matrixinitialize                             *)
(* procedure lengthinitialize                             *)
(* procedure orderinitialize                              *)
(* procedure basischange                                  *)
(* function  scalarproduct                                *)
(* function  equalvectors                                 *)
(* procedure insertroot                                   *)
(* procedure computepositiveroots                         *)
(* function  congruenceclass                              *)
(* procedure computelevellist                             *)
(* procedure computehighestweights                        *)
(* procedure createchain                                  *)
(* function  aorder                                       *)
(* function  border                                       *)
(* function  dorder                                       *)
(* function  eorder                                       *)
(* function  forder                                       *)
(* function  stabilizerorder                              *)
(* function  indexfunction                                *)
(* function  weyldimensionformula                         *)
(* procedure flagsubweights                               *)
(* procedure findxiroots                                  *)
(* procedure reflectdominant                              *)
(* function  previousmultiplicity                         *)
(* function  doublesummation                              *)
(* procedure computemultiplicities                        *)
(* procedure computetable                                 *)
(* procedure printline                                    *)
(* procedure printsegment                                 *)
(* procedure printtable                                   *)
(* multiplicitytable (main block)                         *)
```

```
const

(* the maximum rank of the lie algebra.                          *)

maxrank = 6;

(* the maximum number of representations in the table.           *)

maxrepresentations = 100;

(* the maximum number of multiplicities in the table,            *)
(* i.e. maxrepresentations * (maxrepresentations + 1) /          *)
(* 2.                                                            *)

maxmultiplicities = 5050;

(* the maximum number of columns in each output segment          *)
(* of the table.                                                 *)

maxcolumns = 30;

(* the orders of the weyl groups of algebras e6, e7, e8,         *)
(* f4, and g2.                                                   *)

e6order = 51840;
e7order = 2903040;
e8order = 696729600;
f4order = 1152;
g2order = 12;

(* the number of primes stored in array 'prime', and             *)
(* listed in the 'value' segment.                                *)

numprimes = 100;
```

```
type

nonnegative = 0..maxint;
classicaltypes = (a, bc, d);
rankrange = 1..maxrank;
representationrange = 1..maxrepresentations;
multirange = 1..maxmultiplicities;
classrange = 0..maxrank;
vector = array [rankrange] of integer;
rowtype = array [representationrange] of nonnegative;

(* pointer type and record type used for the positive      *)
(* roots of the algebra. each record of type 'root'        *)
(* contains data about a positive root.                    *)

rootlink = ^root;
root =
  record

  (* the root expressed in the alpha-basis and the         *)
  (* omega-basis.                                          *)

  alpha, omega: vector;

  (* the level of a root is the number of simple roots     *)
  (* which must be subtracted from the highest root in     *)
  (* order to obtain the root.  if the root is a           *)
  (* 'xi-root' (see procedure 'findxiroots'), the          *)
  (* 'orbitsize' is the number of roots (not only          *)
  (* positive roots) in the orbit of the root under the    *)
  (* action of the group generated by the negative of the  *)
  (* identity transformation and the subgroup of the weyl  *)
  (* group which stabilizes the weight for which the       *)
  (* multiplicity is being computed.                       *)

  level, orbitsize: nonnegative;

  nextroot: rootlink;
  end;

(* pointer type and record type used for the dominant      *)
(* weights of the weight system of a given                 *)
(* representation.  each record of type 'weight' contains  *)
(* data about some dominant weight.                        *)

weightlink = ^weight;
weight =
  record

  (* the weight in the omega-basis.                        *)

  omega: vector;

  (* 'multiplicity' is the multiplicity of the weight.     *)
```

```
    (* 'depth' is the least number of positive roots which  *)
    (* must be subtracted from the highest weight of the    *)
    (* representation to obtain the weight.  'level' is the *)
    (* scalar product of the weight with the vector defined *)
    (* by the property tnat its scalar product with every   *)
    (* simple root is 1.                                    *)

    multiplicity, depth, level: nonnegative;

    nextweight: weightlink;
    end;

  (* pointer type and record type used by procedure         *)
  (* 'createchain' and functions 'aorder' through 'forder'  *)
  (* and 'stabilizerorder'.  these procedures compute the   *)
  (* order of the subgroup of the weyl group which          *)
  (* stabilizes a given weight.  each record contains data  *)
  (* about a component of the given weight.                 *)

  link = ^node;
  node =
    record
    left, right: link;
    nonzero: boolean;
    end;
```

```
var

(* the cartan matrix and its inverse.  'inversematrix' is *)
(* multiplied by the determinant of the cartan matrix so  *)
(* that all the entries will be integral.                 *)

cartanmatrix, inversematrix: array [rankrange, rankrange] of
  integer;

(* the determinant of the cartan matrix.                  *)

determinant: nonnegative;

(* the squares of the lengths of the simple roots.        *)

squarelength: vector;

(* the type of the algebra, 'a', 'b', 'c', 'd', 'e', 'f', *)
(* or 'g'.                                                *)

latype: char;

(* the rank of the algebra; the congruence class of       *)
(* representations in the table; the number of            *)
(* representations in the table.                          *)

rank, class, numrepresentations: integer;

(* a pointer to the head node of the list of positive     *)
(* roots.                                                 *)

rootlist: rootlink;

(* an array which holds the list of highest weights.      *)

weightarray: array [representationrange] of weight;

(* an array which holds the weight multiplicities.        *)

multiarray: array [multirange] of nonnegative;

(* arrays which hold data about the corresponding weights *)
(* in 'weightarray'. 'stabarray' contains the order of    *)
(* the stabilizer of the weight. 'orbitsizearray'         *)
(* contains the size of the orbit of the weight under the *)
(* weyl group. 'dimensionarray' contains the dimension of *)
(* the representation with the weight as highest weight.  *)
(* 'productarray' contains the scalar product (weight,    *)
(* weight), i.e. the square length of the weight.         *)
(* 'levelarray' contains the level of the weight.         *)

stabarray, orbitsizearray, dimensionarray, productarray,
  levelarray: rowtype;
```

```
(* an array of pointers of type 'link' used to hold the   *)
(* chain created by procedure 'createchain'.              *)

carrier: array [rankrange] of link;

(* the orders of the weyl groups of the algebras of the   *)
(* classical types a, b, c, and d, and of rank between 1  *)
(* and maxrank.                                           *)

order: array [classicaltypes, rankrange] of nonnegative;

(* an array containing the first 'numprimes' prime        *)
(* numbers; they are needed by procedure                  *)
(* 'weyldimensionformula'.                                *)

prime: array [1..numprimes] of nonnegative;

(* the starting time of the program.                      *)

starttime: integer;
```

```
value

multiarray = (maxmultiplicities of 0);
prime =
  (   2,    3,    5,    7,   11,   13,   17,   19,
     23,   29,   31,   37,   41,   43,   47,   53,
     59,   61,   67,   71,   73,   79,   83,   89,
     97,  101,  103,  107,  109,  113,  127,  131,
    137,  139,  149,  151,  157,  163,  167,  173,
    179,  181,  191,  193,  197,  199,  211,  223,
    227,  229,  233,  239,  241,  251,  257,  263,
    269,  271,  277,  281,  283,  293,  307,  311,
    313,  317,  331,  337,  347,  349,  353,  359,
    367,  373,  379,  383,  389,  397,  401,  409,
    419,  421,  431,  433,  439,  443,  449,  457,
    461,  463,  467,  479,  487,  491,  499,  503,
    509,  521,  523,  541);
```

```
function validinput: boolean;

(* this function checks the validity of the input and      *)
(* returns 'true' iff the input is valid.                  *)

begin (* validinput *)

if ((latype = 'a') and (2 <= rank) and (rank <= 8) and
      (0 <= class) and (class <= rank) or
    (latype = 'b') and (2 <= rank) and (rank <= 8) and
      (0 <= class) and (class <= 1) or
    (latype = 'c') and (2 <= rank) and (rank <= 8) and
      (0 <= class) and (class <= 1) or
    (latype = 'd') and (4 <= rank) and (rank <= 8) and
      (0 <= class) and (class <= 3) or
    (latype = 'e') and (6 <= rank) and (rank <= 8) and
      (0 <= class) and (class <= 8 - rank) or
    (latype = 'f') and (rank = 4) and (class = 0) or
    (latype = 'g') and (rank = 2) and (class = 0)) and
    (0 <= numrepresentations) and
    (numrepresentations <= maxrepresentations)
  then validinput := true
  else validinput := false;

end; (* validinput *)
```

```
procedure speciallinearmatrix;
begin (* speciallinearmatrix *)
end; (* speciallinearmatrix *)

procedure matrixinitialize;
begin (* matrixinitialize *)
end; (* matrixinitialize *)

procedure lengthinitialize;
begin (* lengthinitialize *)
end; (* lengthinitialize *)

procedure orderinitialize;
begin (* orderinitialize *)
end; (* orderinitialize *)

procedure basischange (omega: vector; var alpha: vector);
begin (* basischange *)
end; (* basischange *)

function scalarproduct (vector1, vector2: vector): integer;
begin (* scalarproduct *)
end; (* scalarproduct *)

function equalvectors (vector1, vector2: vector): boolean;
begin (* equalvectors *)
end; (* equalvectors *)

procedure insertroot (root1: rootlink; newroot: vector;
  newlevel: nonnegative);
begin (* insertroot *)
end; (* insertroot *)

procedure computepositiveroots;
begin (* computepositiveroots *)
end; (* computepositiveroots *)
```

```
function congruenceclass (weight: vector): classrange;

(* this function computes and returns the congruence      *)
(* class of 'weight'.                                     *)

var

i: nonnegative;
temp1, temp2: integer;

begin (* congruenceclass *)

case latype of
  'a':
    begin
    temp1 := 0;
    for i := 0 to rank do
      temp1 := temp1 + i * weight [i];
    congruenceclass := temp1 mod (rank + 1);
    end;
  'b':
    congruenceclass := weight [rank] mod 2;
  'c':
    begin
    i := 1;
    temp1 := 0;
    while i <= rank do
      begin
      temp1 := temp1 + weight [i];
      i := i + 2;
      end;
    congruenceclass := temp1 mod 2;
    end;
  'd':
    begin
    temp1 := (weight [rank - 1] + weight [rank]) mod 2;
    temp2 := 0;
    i := 1;
    while i <= rank - 2 do
      begin
      temp2 := temp2 + weight [i];
      i := i + 2;
      end;
    temp2 := 2 * temp2;
    temp2 := temp2 + (rank - 2) * weight [rank - 1] +
      rank * weight [rank];
    temp2 := temp2 mod 4;
    if temp1 = 0
      then if temp2 = 0
             then congruenceclass := 0
             else congruenceclass := 1
      else if temp2 <= 1
             then congruenceclass := 2
             else congruenceclass := 3;
```

```
    end;
  'e':
    case rank of
      6: begin
         temp1 := weight [1] - weight [3] + weight [5] -
           weight [6];
         temp2 := abs (temp1) mod 3;
         if (temp1 >= 0) or (temp2 = 0)
           then congruenceclass := temp2
           else congruenceclass := 3 - temp2;
         end;
      7: congruenceclass :=
           (weight [2] + weight [5] + weight [7]) mod 2;
      8: congruenceclass := 0;
      end;
  'f', 'g':
    congruenceclass := 0;
  end;

end; (* congruenceclass *)
```

```
procedure computelevellist (var lastweight: weightlink;
  inverserowsum: vector; level: nonnegative; var numweights:
  integer);

(* this procedure generates the weights with level          *)
(* 'level', determines which weights are in congruence      *)
(* class 'class', and adds those weights to the end of      *)
(* the weight list ('lastweight' always points to the       *)
(* last node in the current weight list.)  each time a      *)
(* new weight is added to the weight list, 'numweights'     *)
(* is incremented.  the level of a weight is one plus       *)
(* twice the scalar product of 'inverserowsum' with         *)
(* 'weight';  however, in this procedure 'level' refers     *)
(* to the unaltered scalar product.  the set of vectors     *)
(* 'weight' with non-negative integral components such      *)
(* that the scalar procuct of 'weight' with                 *)
(* 'inverserowsum' is 'level' is generated by procedure     *)
(* 'recursion' included within this procedure.              *)

var
weight: vector;
newweight: weightlink;

procedure recursion (sum: nonnegative; index: rankrange);
var
testvalue: nonnegative;
begin (* recursion *)
if index < rank
  then
    (* try all possible nonnegative values for             *)
    (* 'weight [index]'.                                   *)
    for testvalue := 0 to (sum div inverserowsum [index]) do
      begin
      weight [index] := testvalue;
      recursion (sum - testvalue * inverserowsum [index],
        index + 1);
    end
  else
    (* see if 'sum' is a multiple of 'inverserowsum        *)
    (* [rank]'.                                            *)
    if sum mod inverserowsum [rank] = 0
      then
        begin
        (* set 'weight [rank]' to the correct value, and  *)
        (* determine if 'weight' is in the correct        *)
        (* congruence class.                              *)
        weight [rank] := sum div inverserowsum [rank];
        if congruenceclass (weight) = class
          then
            begin
            (* add 'weight' to the list of highest       *)
            (* weights after 'lastweight'.               *)
            new (newweight);
            newweight^.omega := weight;
```

```
                newweight^.level :=
                  (2 * level) div determinant + 1;
                newweight^. nextweight := nil;
                lastweight^. nextweight := newweight;
                lastweight := newweight;
                numweights := numweights + 1;
                end;
        end;
end; (* recursion *)

begin (* computelevellist *)

recursion (level, 1);

end; (* computelevellist *)
```

```
procedure computehighestweights;

(* this procedure first initializes the vector              *)
(* 'inverserowsum' which is used by 'computelevellist' to   *)
(* determine the levels of weights.  it then allocates a    *)
(* dummy head node for the weight list; 'weightlist'        *)
(* points to this head node.  the highest weights are       *)
(* then computed by increasing level by repeated calls to   *)
(* 'computelevellist'.  the weights are then transferred    *)
(* from the dynamic weight list to the array                *)
(* 'weightarray', and finally the procedure disposes of     *)
(* the records in the dynamic weight list.                  *)

var
numweights: integer;
inverserowsum: vector;
i, j: rankrange;
level: nonnegative;
weightlist, lastweight, weight1: weightlink;
column: representationrange;

begin (* computehighestweights *)

(* initialize 'inverserowsum'. *)

for i := 1 to rank do
  begin
  inverserowsum [i] := 0;
  for j := 1 to rank do
    inverserowsum [i] :=
      inverserowsum [i] + inversematrix [i,j];
  end;

(* generate the required number of highest weights, level   *)
(* level, storing them in the list with head pointer        *)
(* 'weightlist'.                                            *)

new (weightlist);
weightlist^. nextweight := nil;
lastweight := weightlist;
numweights := 0;
level := 0;
while numweights < numrepresentations do
  begin
  computelevellist
    (lastweight, inverserowsum, level, numweights);
  level := level + 1;
  end;

(* transfer the first 'numrepresentations' weights in       *)
(* 'weightlist' to 'weightarray'.                           *)

weight1 := weightlist^. nextweight;
for column := 1 to numrepresentations do
```

```
    begin
    weightarray [column] := weight1^;
    weightarray [column]. nextweight := nil;
    weight1 := weight1^. nextweight;
    end;

  (* dispose of the dynamic list of weights. *)

  while weightlist <> nil do
    begin
    weight1 := weightlist^. nextweight;
    dispose (weightlist);
    weightlist := weight1;
    end;

  end; (* computehighestweights *)
```

```
procedure createchain (weight: vector; var first, last:
  link);
begin (* createchain *)
end; (* createchain *)

function aorder (var l1, l2: link): nonnegative;
begin (* aorder *)
end; (* aorder *)

function border (var l1, l2: link): nonnegative;
begin (* border *)
end; (* border *)

function dorder (var l1, l2: link): nonnegative;
begin (* dorder *)
end; (* dorder *)

function eorder (var l1, l2: link): nonnegative;
begin (* eorder *)
end; (* eorder *)

function forder (var l1, l2: link): nonnegative;
begin (* forder *)
end; (* forder *)

function stabilizerorder (weight: vector): nonnegative;
begin (* stabilizerorder *)
end; (* stabilizerorder *)
```

```
function indexfunction (row, column: representationrange):
  multirange;

(* this function computes the 'multiarray' index          *)
(* corresponding to the two coordinates 'row' and         *)
(* 'column' of an entry in the upper-triangular table of  *)
(* weight multiplicities which will be output.            *)

begin (* indexfunction *)

indexfunction := (row - 1) * maxrepresentations + column -
  (row - 1) * row div 2;

end; (* indexfunction *)
```

```
function weyldimensionformula (weight: vector): nonnegative;

(* this function computes the total dimension of the      *)
(* representation with highest weight 'weight', using the *)
(* weyl dimension formula.  in order to allow the         *)
(* computations to remain integral, the running product   *)
(* is stored in terms of its prime factorization in the   *)
(* array 'power'.                                         *)

var

n: 1..numprimes;
numerator, denominator: integer;
count, result: nonnegative;
i: rankrange;
power: array [1..numprimes] of integer;
root1: rootlink;

begin (* weyldimensionformula *)

(* initialize the array of powers.                        *)

for n := 1 to numprimes do
  power [n] := 0;

(* compute the product of the factors, one factor for     *)
(* each positive root.                                    *)

root1 := rootlist;
while root1 <> nil do
  begin

  (* compute the numerator and denominator of this        *)
  (* factor.                                              *)

  numerator := 0;
  for i := 1 to rank do
    numerator := numerator +
      (weight [i] + 1) * root1^. alpha [i] *
      squarelength [i];
  numerator := numerator div 2;
  denominator := 0;
  for i := 1 to rank do
    denominator := denominator +
      root1^. alpha [i] * squarelength [i];
  denominator := denominator div 2;

  (* mulitply by the numerator and divide by the          *)
  (* denominator.                                         *)

  n := 1;
  while numerator <> 1 do
    begin
    if numerator mod prime [n] <> 0
```

```
        then n := n + 1
        else
          begin
          power [n] := power [n] + 1;
          numerator := numerator div prime [n];
          end;
      end;
    n:= 1;
    while denominator <> 1 do
      begin
      if denominator mod prime [n] <> 0
        then n := n + 1
        else
          begin
          power [n] := power [n] - 1;
          denominator := denominator div prime [n];
          end;
      end;

    root1 := root1^. nextroot;
    end;

  (* compute the result.                                    *)

  result := 1;
  for n := 1 to numprimes do
    for count := 1 to power [n] do
      result := result * prime [n];
  weyldimensionformula := result;

  end; (* weyldimensionformula *)
```

```
procedure flagsubweights (repnumber: representationrange);

(* this procedure recomputes the dominant weight system   *)
(* of the representation with highest weight 'weightarray *)
(* [repnumber]'.  for each dominant weight it sets the    *)
(* corresponding element of 'multiarray' to 1, indicating *)
(* to procedure 'computemultiplicities' that a            *)
(* multiplicity must be computed. for more comments on    *)
(* the algorithm, see procedure 'computesubweights'.      *)

var

weight1, weight2: integer;
newweight: vector;
dominant: boolean;
root1: rootlink;
i: rankrange;

begin (* flagsubweights *)

(* set the element of 'multiarray' corresponding to the   *)
(* highest weight to 1: the multiplicity of the highest   *)
(* weight.                                                *)

multiarray [indexfunction (repnumber, repnumber)] := 1;

(* compute the other dominant weights of the              *)
(* representation. 'weight1' moves through the weight     *)
(* array.                                                 *)

weight1 := repnumber;
while weight1 <> 0 do
  begin
  if multiarray [indexfunction (weight1, repnumber)] = 1
    then
      begin

      (* 'root1' links through the root list.             *)

      root1 := rootlist;
      while root1 <> nil do
        begin

        (* subtract the root 'root1' from the weight      *)
        (* 'weightarray [weight1]'.                       *)

        for i := 1 to rank do
          newweight [i] := weightarray [weight1]. omega [i]
            - root1^. omega [i];

        (* determine whether 'newweight' is dominant.     *)

        dominant := true;
        for i := 1 to rank do
```

```
            dominant := dominant and (newweight [i] >= 0);
          if dominant
            then
              begin

              (* 'newweight' is dominant and hence is a      *)
              (* dominant weight of the representation;      *)
              (* find it in 'weightarray' and set the        *)
              (* corresponding element of 'weightarray' to   *)
              (* 1.                                          *)

              weight2 := repnumber - 1;

              while weight2 >= 1 do
                if equalvectors (weightarray [weight2]. omega,
                  newweight)
                  then
                    begin
                    multiarray
                      [indexfunction (weight2, repnumber)] :=
                      1;
                    weight2 := 0;
                    end
                  else weight2 := weight2 - 1;

              end;

          root1 := root1^. nextroot;
          end;
        end;
    weight1 := weight1 - 1;
    end;

end; (* flagsubweights *)
```

```
procedure findxiroots (subweight: vector; subgrouporder:
  nonnegative);
begin (* findxiroots *)
end; (* findxiroots *)

procedure reflectdominant (factor1: vector; var
  dominant1: vector);
begin (* reflectdominant *)
end; (* reflectdominant *)
```

```
function previousmultiplicity (dominant1: vector; repnumber:
  representationrange): nonnegative;

(* this function first searches in the array of dominant  *)
(* weights for the weight 'dominant1'. if 'dominant1' is *)
(* found, the function returns its multiplicity in the    *)
(* representation with highest weight 'weightarray        *)
(* [repnumber]' (which multiplicity will have been        *)
(* previously computed); otherwise, the function returns  *)
(* 0 (since if 'dominant1' is not found its level is      *)
(* higher than that of 'weightarray [repnumber]' and      *)
(* hence its multiplicity is 0).                          *)

var

found: boolean;
weight1: representationrange;

begin (* previousmultiplicity *)

weight1 := repnumber;
found := false;
while (weight1 > 1) and not found do
  begin
  found :=
    equalvectors (weightarray [weight1]. omega, dominant1);
  if not found
    then weight1 := weight1 - 1;
  end;
if found
  then previousmultiplicity :=
         multiarray [indexfunction (weight1, repnumber)]
  else previousmultiplicity := 0;

end; (* previousmultiplicity *)
```

```
function doublesummation (subweight: vector; repnumber:
  representationrange): nonnegative;

(* this function computes and returns the value of the    *)
(* double summation in the modified freudenthal formula.  *)
(* the outer sum is over the orbits of the root system;   *)
(* this sum is implemented by searching through the root  *)
(* list for the xi-roots (the orbit representatives).  if *)
(* a xi-root is encountered, the inner sum is computed;   *)
(* it is a finite sum of terms of the form scalar product *)
(* times multiplicity.  as soon as a zero term is         *)
(* encountered in the inner sum, we know that all further *)
(* terms are zero. instead of using the actual            *)
(* multiplicity of 'factor1', which is not available, we  *)
(* reflect 'factor1' into the dominant chamber using      *)
(* procedure 'reflectdominant', and then call function    *)
(* 'previousmultiplicity' to find the multiplicity of the *)
(* weight 'dominant1' computed by 'reflectdominant'. note *)
(* that since 'dominant1' is a dominant weight of higher  *)
(* level than 'subweight', the multiplicity of            *)
(* 'dominant1' will have been computed by the time we     *)
(* call 'previousmultiplicity'.                           *)

var

root1: rootlink;
innerterm, innersum, result: integer;
count: nonnegative;
i: rankrange;
factor1, dominant1: vector;

begin (* doublesummation *)

result := 0;
root1 := rootlist;
while root1 <> nil do
  begin
  if root1^. orbitsize <> 0
    then
      begin
      count := 0;
      innersum := 0;
      repeat
        count := count + 1;
        for i := 1 to rank do
          factor1 [i] := subweight [i] +
            count * root1^. omega [i];
        reflectdominant (factor1, dominant1);
        innerterm :=
          previousmultiplicity (dominant1, repnumber);
        if innerterm <> 0
          then innerterm := innerterm *
                 scalarproduct (factor1, root1^. alpha);
      innersum := innersum + innerterm;
```

```
        until innerterm = 0;
        result := result + root1^. orbitsize * innersum;
        end;
    root1 := root1^. nextroot;
    end;
doublesummation := result;

end; (* doublesummation *)
```

```
procedure computemultiplicities (repnumber:
  representationrange);

(* this procedure computes the multiplicities of t      *)
(* dominant weights in the representation with highest  *)
(* weight 'weightarray [repnumber]'.  we know that the  *)
(* multiplicity of the highest weight is one, and from  *)
(* this the multiplicities of weights with lower levels *)
(* are computed recursively using the modified          *)
(* freudenthal formula.  note that in order to obtain the *)
(* correct multiplicity for a given weight, we must     *)
(* divide the value computed by function                *)
(* 'doublesummation' by the difference of 'highproduct' *)
(* and 'subproduct'.  the total dimension of the        *)
(* representation is also computed (based on the        *)
(* multiplicities of the dominant weights and the sizes *)
(* of their orbits) and stored in 'dimension'.  this is *)
(* done so that the computations can be checked by      *)
(* comparing the result obtained by this method with the *)
(* result given by the weyl dimension formula.          *)

var

highproduct, subproduct, dimension: nonnegative;
i: rankrange;
alpha, omega: vector;
multiindex: multirange;
weight1: representationrange;

begin (* computemultiplicities *)

(* compute 'highproduct'.                               *)

for i := 1 to rank do
  omega [i] := weightarray [repnumber]. omega [i] + 1;
basischange (omega, alpha);
highproduct := scalarproduct (omega, alpha);

(* compute the multiplicities of the dominant weights.  *)

dimension := orbitsizearray [repnumber];
for weight1 := repnumber - 1 downto 1 do
  begin
  multiindex := indexfunction (weight1, repnumber);
  if multiarray [multiindex] = 1
    then
      begin

      (* compute 'subproduct'.                          *)

      for i := 1 to rank do
        omega [i] := weightarray [weight1]. omega [i] + 1;
      basischange (omega, alpha);
      subproduct := scalarproduct (omega, alpha);
```

```
        (* compute the muliplicity and increment                    *)
        (* 'dimension'.                                              *)

        findxiroots
          (weightarray [weight1]. omega, stabarray [weight1]);
        multiarray [multiindex] :=
          doublesummation
            (weightarray [weight1]. omega, repnumber)
          div (highproduct - subproduct);
        dimension := dimension +
          multiarray [multiindex] * orbitsizearray [weight1];

        end;
    end;

  (* check the dimension.                                           *)

  if dimension <> dimensionarray [repnumber]
    then writeln ('dimensions not equal for representation',
          repnumber: 3);

  end; (* computemultiplicities *)
```

```
procedure computetable;

(* this procedure computes the information to be inoluded *)
(* in the output:  the contents of the arrays             *)
(* 'stabarray', 'orbitsizearray', 'productarray',         *)
(* 'dimensionarray', and 'levelarray', as well as the     *)
(* multiplicites in 'multiarray'.                         *)

var

alpha: vector;
repnumber: representationrange;
weylgrouporder: nonnegative;

begin (* computetable *)

(* determine 'weylgrouporder', the order of the weyl      *)
(* group of the algebra.                                  *)

case latype of
  'a':  weylgrouporder := order [a, rank];
  'b', 'c': weylgrouporder := order [bc, rank];
  'd':  weylgrouporder := order [d, rank];
  'e':  case rank of
          6: weylgrouporder := e6order;
          7: weylgrouporder := e7order;
          8: weylgrouporder := e8order;
        end;
  'f':  weylgrouporder := f4order;
  'g':  weylgrouporder := g2order;
  end;

(* compute the information to be output.                  *)

for repnumber := 1 to numrepresentations do
  begin
  stabarray [repnumber] :=
    stabilizerorder (weightarray [repnumber]. omega);
  orbitsizearray [repnumber] :=
    weylgrouporder div stabarray [repnumber];
  basischange (weightarray [repnumber]. omega, alpha);
  productarray [repnumber] :=
    scalarproduct (weightarray [repnumber]. omega, alpha);
  dimensionarray [repnumber] :=
    weyldimensionformula (weightarray [repnumber]. omega);
  levelarray [repnumber] := weightarray [repnumber]. level;
  flagsubweights (repnumber);
  computemultiplicities (repnumber);
  end;

end; (* computetable *)
```

```
procedure printline (tablerow: rowtype; left, right, row:
  representationrange; inmultis: boolean);

(* this procedure outputs the row of the table contained  *)
(* in 'tablerow' between columns 'left' and 'right'       *)
(* inclusive.  if 'tablerow' is a row of multiplicities,  *)
(* 'row' is the (ordinal) number of the row; for other    *)
(* rows, 'row' is 1.  'inmultis' is true iff 'tablerow'   *)
(* is a row of multiplici- ties.  in order to enforce a   *)
(* standard width, entries in the row are split into      *)
(* 'pieces' (i.e. three-digit segments), starting with    *)
(* the lowest three digits, and printed as a column of    *)
(* pieces.  all of the computation done by this procedure *)
(* is necessary to ensure correct formatting of the       *)
(* output.                                                *)

var

innumber: array [representationrange] of boolean;
divisor, maximum, numlines, line: nonnegative;
column: representationrange;
piece: 0..999;

begin (* printline *)

(* find the maximum entry in tablerow.                    *)

maximum := 0;
for column := 1 to numrepresentations do
  if tablerow [column] > maximum
    then maximum := tablerow [column];

(* find the number of lines needed to output this row,    *)
(* i.e. the number of 'pieces' in 'maximum'.              *)

numlines := 0;
while maximum <> 0 do
  begin
  maximum := maximum div 1000;
  numlines := numlines + 1;
  end;
if numlines = 0
  then numlines := 1;

(* initialize 'innumber' and 'divisor'.  'innumber        *)
(* [column]' is true iff the leftmost piece of 'tablerow  *)
(* [column]' has been reached. 'divisor' defines the      *)
(* current piece being output.                            *)

for column := left to right do
  innumber [column] := false;
divisor := 1;
for line := 1 to numlines - 1 do
  divisor := divisor * 1000;
```

```
(* output this row of the table.                                    *)

for line := 1 to numlines do
  begin
  for column := left to right do
    begin
    piece := tablerow [column] div divisor;
    if innumber [column]
      then
        if piece > 99
          then write (' ', piece: 3)
          else
            if piece > 9
              then write (' 0', piece: 2)
              else write (' 00', piece: 1)
      else
        if piece = 0
          then
            if line <> numlines
              then write ('    ')
              else if inmultis
                     then
                       if column < row
                         then write ('    ')
                         else write ('   .')
                     else write ('   0')
          else
            begin
            innumber [column] := true;
            if piece > 99
              then write (' ', piece: 3)
              else
                if piece > 9
                  then write ('  ', piece: 2)
                  else write ('   ', piece: 1);
            end;
    tablerow [column] :=
      tablerow [column] - piece * divisor;
    end;
  if line <> numlines
    then writeln;
  divisor := divisor div 1000;
  end;

end; (* printline *)
```

```
procedure printsegment (left, right: representationrange;
  titles: boolean);

(* this procedure prints the segment of the table between *)
(* columns 'left' and 'right' inclusive.  labels for the  *)
(* rows will be printed at the right margin of the        *)
(* segment iff 'titles' is true.                          *)

var

column, row: representationrange;
i: rankrange;
tablerow: rowtype;
hyphen: nonnegative;

begin (* printsegment *)

(* output the scalar products, orbit sizes, levels, and   *)
(* dimensions.                                            *)

row := 1;
printline (productarray, left, right, row, false);
if titles
  then writeln ('  s.p.')
  else writeln;
writeln;
printline (orbitsizearray, left, right, row, false);
if titles
  then writeln ('  o.s.')
  else writeln;
writeln;
printline (levelarray, left, right, row, false);
if titles
  then writeln ('  level')
  else writeln;
writeln;
printline (dimensionarray, left, right, row, false);
if titles
  then writeln ('  dim''n')
  else writeln;
writeln;
write (' ');
for hyphen := 1 to 4 * (right - left + 1) - 1 do
  write ('-');
if titles
  then for hyphen := 1 to 8 do write ('-');
writeln;
writeln;

(* output the highest weights formatted according to the  *)
(* dynkin diagram.                                        *)

for column := left to right do
  write ('  ', weightarray[column].omega[1]: 2);
```

```
writeln;
case latype of
  'a', 'b', 'c', 'f', 'g':
    begin
    for i := 2 to rank do
      begin
      if ((latype = 'b') and (i = rank)) or ((latype = 'f')
        and (i = 3))
        then
          begin
          for column := left to right do
            write ('   >');
          writeln;
          end;
      if (latype in ['c', 'g']) and (i = rank)
        then
          begin
          for column := left to right do
            write ('   <');
          writeln;
          end;
      for column := left to right do
        write ('   ', weightarray [column]. omega [i]: 2);
      if i <> rank
        then writeln;
      end;
    end;
  'd':
    begin
    for i := 2 to rank - 3 do
      begin
      for column := left to right do
       write ('   ', weightarray [column]. omega [i]: 2);
      writeln;
      end;
    for column := left to right do
      write (weightarray [column]. omega [rank]: 2,
             weightarray [column]. omega [rank - 2]: 2);
    writeln;
    for column := left to right do
      write
        ('   ', weightarray [column]. omega [rank - 1]: 2);
    end;
  'e':
    begin
    for column := left to right do
      write ('   !', weightarray [column]. omega [3]: 2);
    writeln;
    for column := left to right do
      write (weightarray [column]. omega [2]: 2,
             weightarray [column]. omega [4]: 2);
    writeln;
    for i := 5 to rank do
      begin
```

```
        for column := left to right do
          write ('  ', weightarray [column]. omega [i]: 2);
        if i <> rank
          then writeln;
        end;
      end;
  end;
if titles
  then writeln ('  weight')
  else writeln;
writeln;
write (' ');
for hyphen := 1 to 4 * (right - left + 1) - 1 do
  write ('-');
if titles
  then for hyphen := 1 to 8 do write ('-');
writeln;
writeln;

(* output the numbers of the columns.                    *)

for column := left to right do
  write ((column - 1): 4);
if titles
  then writeln ('  number')
  else writeln;
writeln;
write (' ');
for hyphen := 1 to 4 * (right - left + 1) - 1 do
  write ('-');
if titles
  then for hyphen := 1 to 8 do write ('-');
writeln;
writeln;

(* output the multiplicities.                            *)

for row := 1 to right do
  begin
  for column := 1 to row - 1 do
    tablerow [column] := 0;
  for column := row to numrepresentations do
    tablerow [column] :=
      multiarray [indexfunction (row, column)];
  printline (tablerow, left, right, row, true);
  if titles
    then writeln ('      ', (row - 1): 3)
    else writeln;
  writeln;
  end;

end; (* printsegment *)
```

```
procedure printtable;

(* this procedure prints the title for the table,          *)
(* determines how the table will be divided into vertical  *)
(* segments, and then calls procedure 'printsegment' to    *)
(* print the segments.                                     *)

var

numsegments, segment: nonnegative;
hyphen: nonnegative;

begin (* printtable *)

(* output the title and compute the number of segments.    *)

writeln (' algebra ', latype, rank: 1, ', representations ',
         'of class ', class: 1);
write (' ');
for hyphen := 1 to 4 * numrepresentations + 7 do
  write ('-');
writeln;
writeln;
numsegments := (numrepresentations - 1) div maxcolumns + 1;

(* output the segments starting with the leftmost..        *)

for segment := 1 to numsegments - 1 do
  begin
  printsegment ((segment - 1) * maxcolumns + 1,
                segment * maxcolumns, true);
  writeln ('1');
  writeln;
  writeln;
  writeln;
  end;
printsegment ((numsegments - 1) * maxcolumns + 1,
              numrepresentations, true);

end; (* printtable *)
```

```
begin (* multiplicitytable *)

starttime := clock;
reset (input);
linelimit (output, maxint);
read (latype, rank, class, numrepresentations);
if validinput
  then
    begin
    matrixinitialize;
    lengthinitialize;
    orderinitialize;
    computepositiveroots;
    computehighestweights;
    computetable;
    printtable;
    end;
writeln
  ('1total time (milliseconds) = ', (clock - starttime): 1);

end. (* multiplicitytable *)
```

Examples of output

The envelope attached to the inside back cover contains tables of the multiplicities for the first 52 representations in each congruence class of the Lie algebras E6, E7, E8, F4, and G2. (Note that E6 and E7 have three and two congruence classes respectively, whereas E8, F4, and G2 each have one.) These tables (except E6 class 2) as well as similar tables for algebras of types A, B, C, and D and ranks up to 8 are included in Bremner, Moody, Patera (to appear).

Time and storage requirements

This section presents the time requirements (in central-processor seconds) and storage requirements (in 60-bit words, in octal) on a Cyber 835 of MULTIPLICITYTABLE for the first 52 representations in each congruence class of each simple Lie algebra of rank up to 8. (For type A just under half of the congruence classes have been omitted owing to the symmetry of the Dynkin diagram.) Time requirements are on the first page and storage requirements are on the second page.

A2	0	26.2	B2	0	40.8	D4	0	41.4	E6	0	66.5
	1	26.6		1	43.9		1	44.1		1	72.7
A3	0	30.1	B3	0	52.2		2	43.7		2	72.2
	1	30.4		1	60.0		3	44.1	E7	0	91.6
	2	30.1	B4	0	60.0	D5	0	50.1		1	101.5
A4	0	32.3		1	71.7		1	50.3	E8	0	146.8
	1	32.3	B5	0	64.5		2	54.7			
	2	32.4		1	79.1		3	54.5	F4	0	95.9
A5	0	35.5	B6	0	71.4	D6	0	52.6			
	1	35.9		1	86.3		1	56.9	G2	0	61.9
	2	35.2	B7	0	77.2		2	59.7			
	3	34.4		1	96.0		3	58.8			
A6	0	36.2	B8	0	63.1	D7	0	56.2			
	1	35.4		1	101.9		1	60.2			
	2	36.4					2	71.9			
	3	36.6	C3	0	55.3		3	70.5			
A7	0	36.6		1	56.9	D8	0	68.0			
	1	37.7	C4	0	63.8		1	65.8			
	2	39.6		1	69.1		2	77.6			
	3	40.2	C5	0	70.0		3	79.3			
	4	41.9		1	74.6						
A8	0	41.4	C6	0	75.6						
	1	40.9		1	81.6						
	2	42.7	C7	0	81.2						
	3	41.4		1	83.6						
	4	43.3	C8	0	94.1						
				1	92.1						

A2 0	36137	B2 0	36045	D4 0	36376	E6 0	37311
1	36122	1	36045	1	36376	1	37343
A3 0	36212	B3 0	36340	2	36376	2	37360
1	36227	1	36340	3	36376	E7 0	40410
2	36212	B4 0	36533	D5 0	36600	1	40410
A4 0	36412	1	36533	1	36664	E8 0	43513
1	36375	B5 0	36761	2	36664		
2	36375	1	36761	3	36664	F4 0	37721
A5 0	36653	B6 0	37326	D6 0	37170		
1	36520	1	37326	1	37136	G2 0	37075
2	36466	B7 0	37743	2	37136		
3	36552	1	37743	3	37121		
A6 0	37037	B8 0	40462	D7 0	37544		
1	37022	1	40462	1	37512		
2	37005			2	37561		
3	37005	C3 0	36303	3	37544		
A7 0	37232	1	36306	D8 0	40237		
1	37077	C4 0	36550	1	40222		
2	37131	1	36501	2	40205		
3	37163	C5 0	36776	3	40205		
4	37131	1	36761				
A8 0	37503	C6 0	37326				
1	37434	1	37360				
2	37402	C7 0	37760				
3	37417	1	37726				
4	37417	C8 0	40445				
		1	40430				

Appendix

A worked example of the fast recursion formula

This Appendix presents an example of hand computation using the fast recursion formula. We consider the representation of D5 with highest weight 0 0 0 1 2; this representation is in congruence class 2 and has dimension 1440. The dominant weights and their multiplicities are

weight	multiplicity
0 0 0 1 2	1
0 0 1 1 0	1
0 1 0 0 1	3
1 0 0 1 0	6
0 0 0 0 1	15

We show how this information may be computed manually using the dominant weight algorithm and the modified Freudenthal formula of Moody, Patera (1982).

We first collect some data needed for the computation. The twenty positive roots of D5 (in the omega-basis) are as follows:

number	root
1	0 1 0 0 0
2	1 -1 1 0 0
3	-1 0 1 0 0
4	1 0 -1 1 1
5	-1 1 -1 1 1
6	1 0 0 -1 1
7	1 0 0 1 -1
8	0 -1 0 1 1
9	1 1 0 -1 1
10	-1 1 0 1 -1
11	1 0 1 -1 -1
12	0 -1 1 -1 1
13	0 -1 1 1 -1
14	-1 1 1 -1 -1
15	1 1 -1 0 0
16	0 0 -1 0 2
17	0 -1 2 -1 -1
18	0 0 -1 2 0

```
 19    -1  2 -1  0  0
 20     2 -1  0  0  0
```

The inverse of the Cartan matrix (scaled up by 4) is

```
(4   4   4   2   2)
(4   8   8   4   4)
(4   8  12   6   6)
(2   4   6   5   3)
(2   4   6   3   5)
```

Hence the inverse-row-sum vector (also scaled up by 4) is (16, 28, 36, 20, 20).

We compute the dominant weights starting from the highest weight by subtracting all positive roots from all previously-computed dominant weights and only keeping the dominant weights obtained at each stage. This gives the following table:

depth	weight	number	
0	0 0 0 1 2	0	(highest weight)
1	0 1 0 0 1	1	(weight-0 minus root-8)
1	0 0 1 1 0	2	(weight-0 minus root-16)
2	0 0 0 0 1	3	(weight-1 minus root-1)
2	1 0 0 1 0	4	(weight-1 minus root-9)

(Weight-4 can also be generated as weight-2 minus root-3.) At depth 3 we compute weight-4 minus root-7 but this is weight-3 at depth 2. Thus the above list is the complete set of dominant weights of this representation.

Using the inverse-row-sum vector we can determine the levels of the weights. The level of a weight is the ordinary scalar product of the weight with the inverse-row-sum vector given above, scaled down by 4 (the determinant of the Cartan matrix), then multiplied by 2 and incremented by 1:

weight					number	level
0	0	0	1	2	0	31
0	1	0	0	1	1	25
0	0	1	1	0	2	29
0	0	0	0	1	3	11
1	0	0	1	0	4	19

Then if we sort and renumber the weights by decreasing level we obtain:

weight					number	level
0	0	0	1	2	0	31
0	0	1	1	0	1	29
0	1	0	0	1	2	25
1	0	0	1	0	3	19
0	0	0	0	1	4	11

In what follows we use this second numbering of the weights.

Before starting the multiplicity computation, we determine the constants {(h+d,h+d)-(g+d,g+d)} which occur in the multiplicity formula (see p. 23), where h = weight-0 is the highest weight, d is the sum of the fundamental weights (equivalently half the sum of the positive roots), and g = weight-i for $1 \leq i \leq 4$. To compute the scalar product (induced by the Killing form, not the ordinary scalar product) recall that the alpha-basis expression for a vector is the transpose of the inverse Cartan matrix times the omega-basis expression, and that the scalar product of two vectors, one in the alpha-basis and one in the omega-basis, is the ordinary scalar product weighted by the square-lengths of the simple roots. Since in type D the Cartan matrix is symmetric and all the simple roots have the same length, to compute the scalar product of two vectors, both in the omega-basis, we merely evaluate the product (vector 1) times (inverse Cartan matrix) times (vector 2).

(We use the inverse Cartan matrix as given at the beginning of this section and ignore scaling factors, since they cancel out in the multiplicity formula.) We then have the following table:

weight	number	(h+d,h+d)-(g+d,g+d)
0 0 1 1 0	1	16
0 1 0 0 1	2	40
1 0 0 1 0	3	72
0 0 0 0 1	4	112

To compute the multiplicity of weight-1 (0 0 1 1 0) we first write down the corresponding xi-roots and their orbit-sizes. (Complete tables of the xi-roots and their orbit sizes for all simple Lie algebras of ranks up to 8 and all possible weights are included in Bremner, Moody, Patera (to appear).)

number	xi-root	orbit-size
1	0 1 0 0 0	6
2	1 0 -1 1 1	12
3	1 0 0 -1 1	12
4	1 1 -1 0 0	6
5	0 0 -1 0 2	2
6	0 0 -1 2 0	2

From the fast recursion formula we then determine that

16(multiplicity-1) =

{6[0] + 12[0] + 12[0] + 6[0] + 2[1(8)] + 2[0]}

and hence that multiplicity-1 equals 1. Here all the inner sums on the right-hand side are zero except for the fifth, where the highest weight occurs (as weight-1 plus xi-root-5); its multiplicity is 1 and its scalar product with xi-root-5 is 8.

For weight-2 (0 1 0 0 1), the xi-roots and their orbit-sizes are

number	xi-root					orbit-size
1	0	1	0	0	0	2
2	-1	-1	1	0	0	12
3	1	0	0	1	-1	12
4	0	-1	0	1	1	6
5	0	-1	-1	1	-1	6
6	2	-1	0	0	0	2

We then determine that

40(multiplicity-2) =

{2[0] + 12[0] + 12[0] + 6[1(12)] + 6[1(8)] + 2[0]},

and hence multiplicity-2 equals 3. In the fourth inner sum the highest weight occurs as weight-2 plus xi-root-4; its multiplicity is 1 and its scalar product with xi-root-4 is 12. In the fifth inner sum weight-1 occurs as weight-2 plus xi-root-5; its multiplicity is 1 and its scalar product with xi-root-5 is 8.

For weight-3 (1 0 0 1 0) the xi-roots and their orbit-sizes are

number	xi-root					orbit-size
1	0	1	0	0	0	8
2	-1	0	1	0	0	12
3	1	0	0	-1	1	8
4	-1	1	0	-1	1	12

We then determine that

72(multiplicity-3) =

{8[0] + 12[1(12)] + 8[0] + 12[3(8)]},

and hence multiplicity-3 equals 6. In the second inner sum weight-1 occurs as weight-3 plus xi-root-2; its multiplicity is 1 and its scalar product with xi-root-2 is 12. In the fourth inner sum, weight-2 occurs as weight-3 plus xi-root-4; its multiplicity is 3 and its scalar product with xi-root-4 is 8.

For weight-4 (0 0 0 0 1) the xi-roots and their orbit-sizes are

number	xi-root	orbit-size
1	0 1 0 0 0	20
2	1 0 0 1 -1	20

We then determine that

112(multiplicity-4) =

{20[3(12)] + 20[6(8)]},

and hence multiplicity-4 equals 15. In the first inner sum weight-2 occurs as weight-4 plus xi-root-1; its multiplicity is 3 and its scalar product with xi-root-1 is 12. In the second inner sum, weight-3 occurs as weight-4 plus xi-root-2; its multiplicity is 6 and its scalar product with xi-root-2 is 8.

References

V. K. Agrawala, J. G. Belinfante (1969), Weight diagrams for Lie group representations: A computer implementation of Freudenthal's algorithm in ALGOL and FORTRAN, BIT 9, 301-314.

R. E. Beck, B. Kolman (1972), A computer implementation of Freudenthal's multiplicity formula, Indag. Math. 34, 350-352.

J. G. F. Belinfante, B. Kolman (1972), A Survey of Lie Groups and Lie Algebras with Applications and Computational Methods, SIAM, Philadelphia.

N. Bourbaki (1968), Groupes et algèbres de Lie, chap. 4, 5 et 6, Hermann, Paris.

M. R. Bremner, R. V. Moody, J. Patera (to appear), Tables of Dominant Weight Multiplicities in Representations of Simple Lie Algebras, Marcel Dekker, New York.

E. B. Dynkin (1957), Maximal subgroups of the classical groups. Supplement: Survey of the basic concepts and facts in the theory of linear representations of semisimple Lie algebras, Amer. Math. Soc. Trans., series 2, vol. 6, 319-378.

H. Freudenthal (1954), Zur Berechnung der Charaktere der halbeinfachen Lieschen Gruppen, I, Indag. Math., 16, 369-376.

I. N. Herstein (1975), Topics in Algebra, second edition, John Wiley, New York.

J. E. Humphreys (1972), Introduction to Lie Algebras and Representation Theory, Springer-Verlag, New York.

N. Jacobson (1974), Basic Algebra I, W. H. Freeman, San Francisco.

---------- (1962), Lie Algebras, Dover, New York.

B. Kolman, R. E. Beck (1973a), Computers in Lie algebras. I: Calculation of inner multiplicities, SIAM J. Appl. Math., 25, 2, 300-312.

---------- (1973b), Freudenthal's inner multiplicity formula, Comp. Phys. Comm., 6, 1, 24-29.

M. I. Krusemeyer (1971), Determining multiplicities of dominant weights in irreducible Lie algebra representations using a computer, BIT 11, 310-316.

F. Lemire, J. Patera (1980), Congruence number, a generalization of SU(3) triality, J. Math. Phys., 21, 8, 2026-2027.

E. M. Loebl (1968), Group Theory and its Applications, vol. I, Academic Press, New York; see also vol. II (1971) and vol. III (1975).

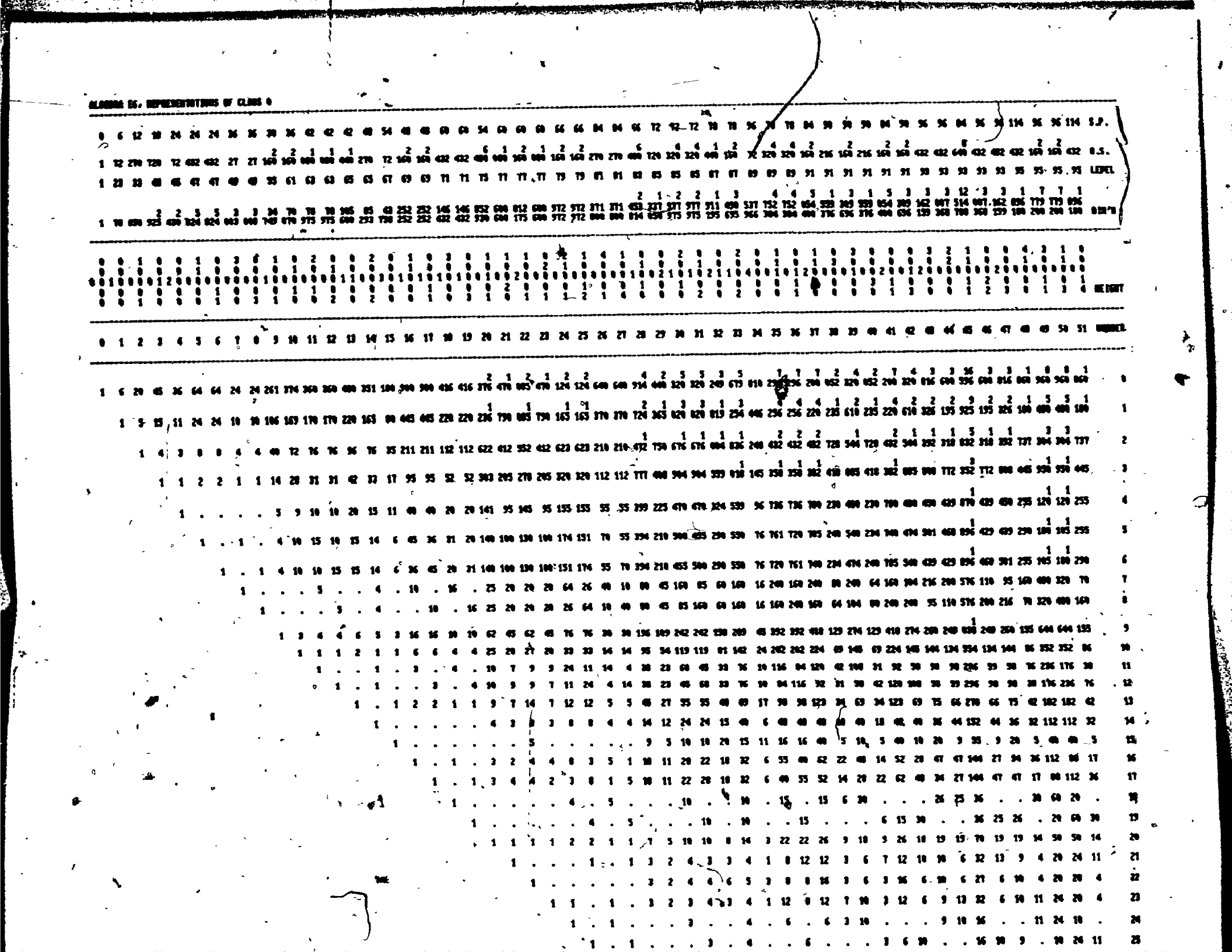
W. McKay, J. Patera, D. Sankoff (1977), The computation of branching rules for representations of semisimple Lie algebras, in R. E. Beck, B. Kolman (eds.), Computers in Non-Associative Rings and Algebras, Academic Press, New York.

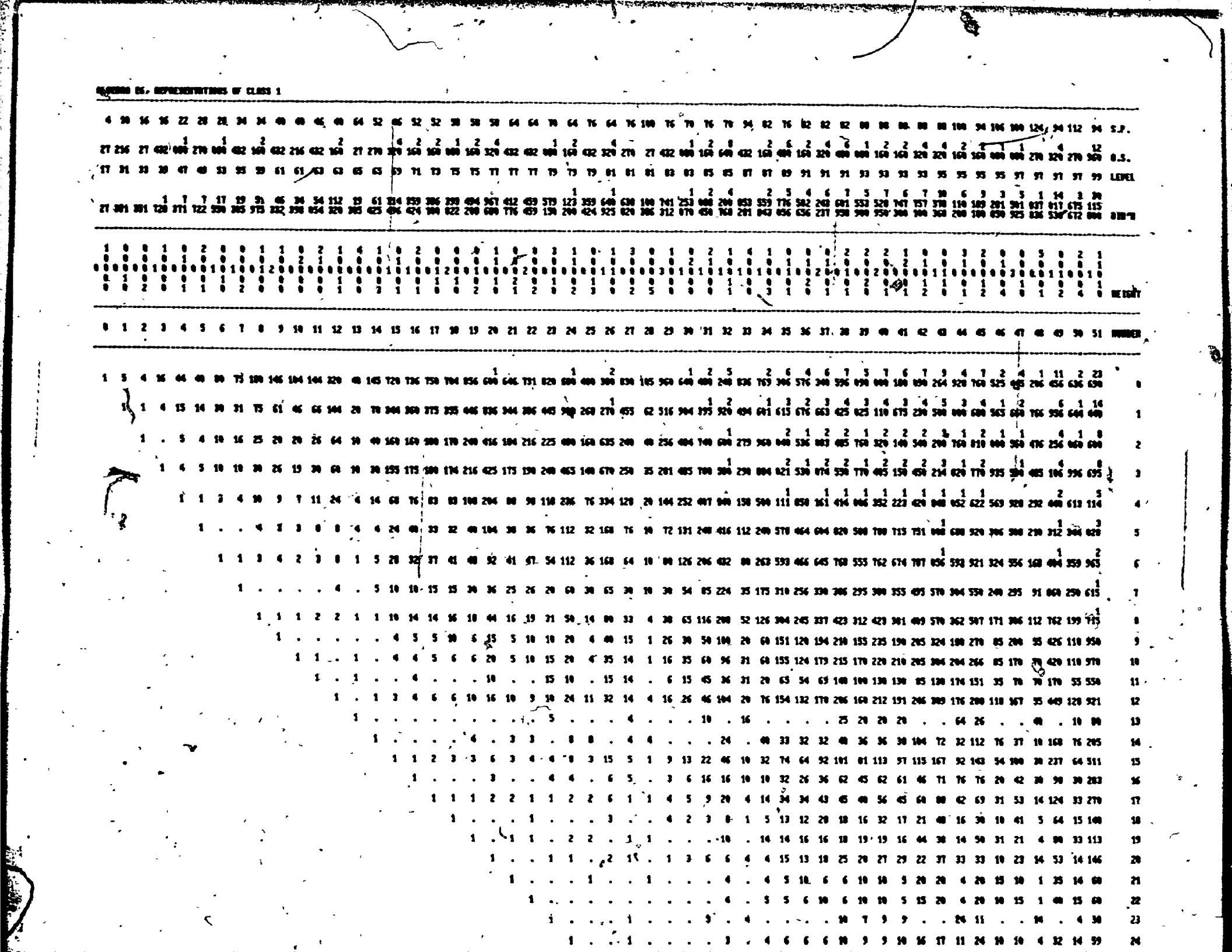
R. V. Moody, J. Patera (1982), Fast recursion formula for weight multiplicities, Bull. Amer. Math. Soc., 7, 1, 237-242.

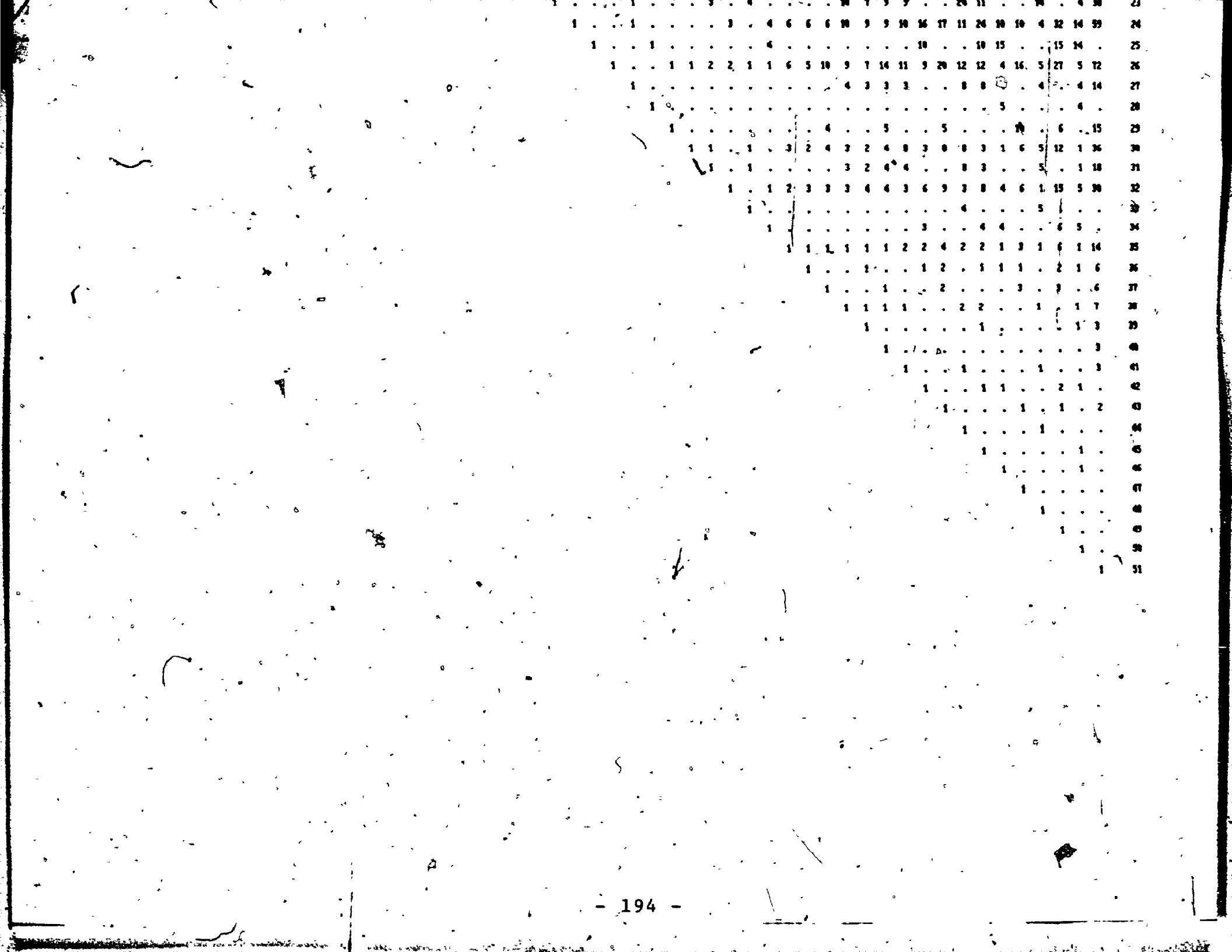
H. Weyl (1950), The Theory of Groups and Quantum Mechanics, Dover, New York.

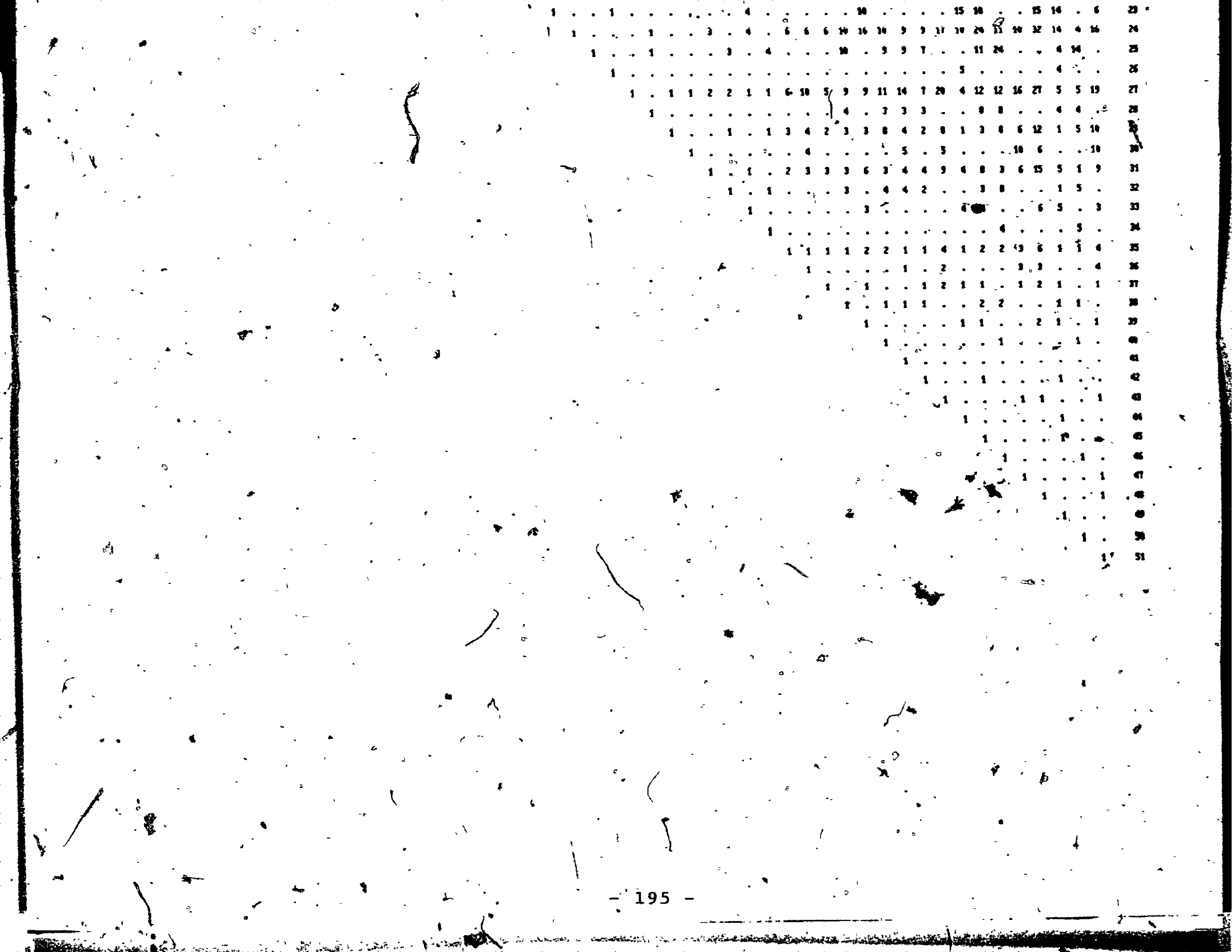
E. P. Wigner (1959), Group Theory and its Application to the Quantum Mechanics of Atomic Spectra, Academic Press, New York.

ALGEBRA 55: REPRESENTATIONS OF CLASS 0



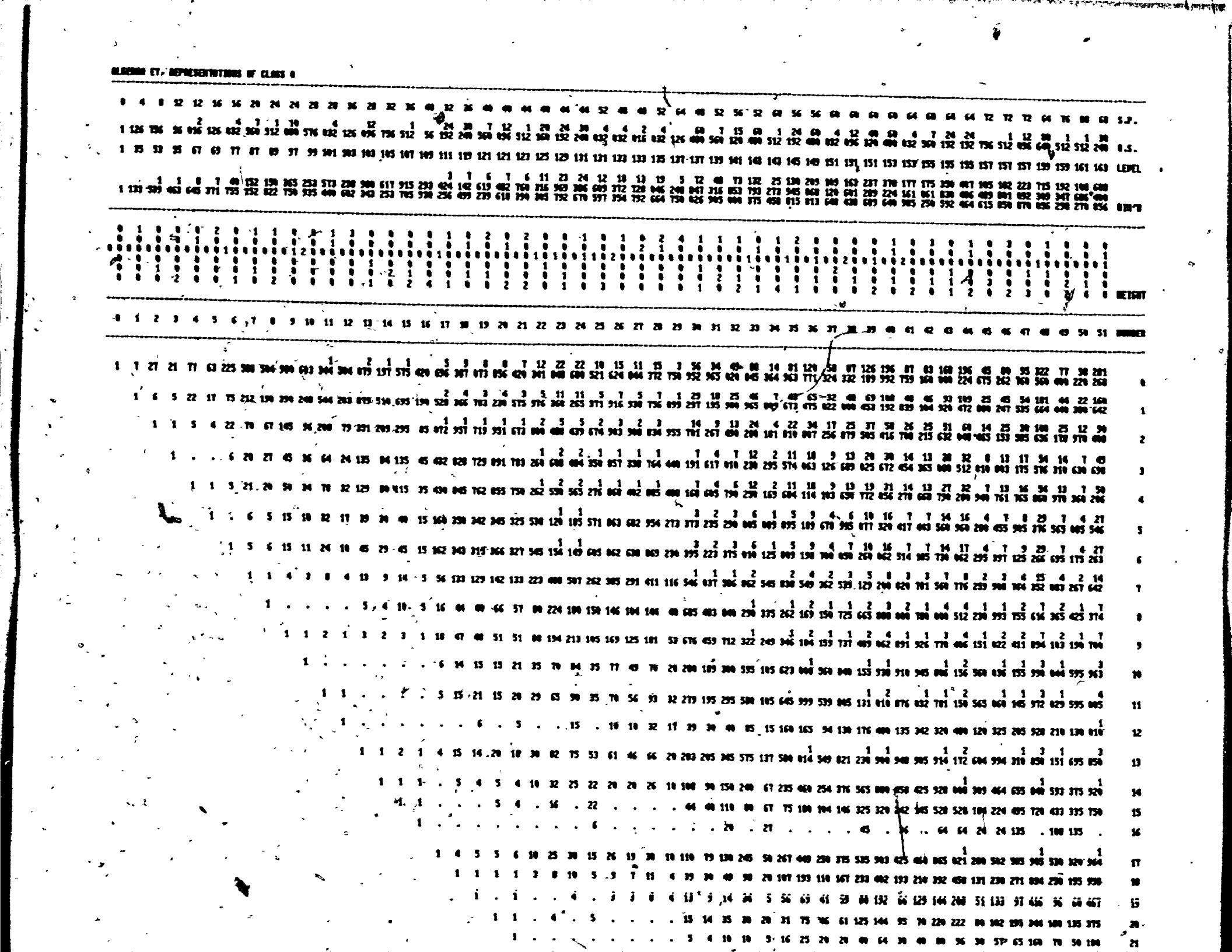


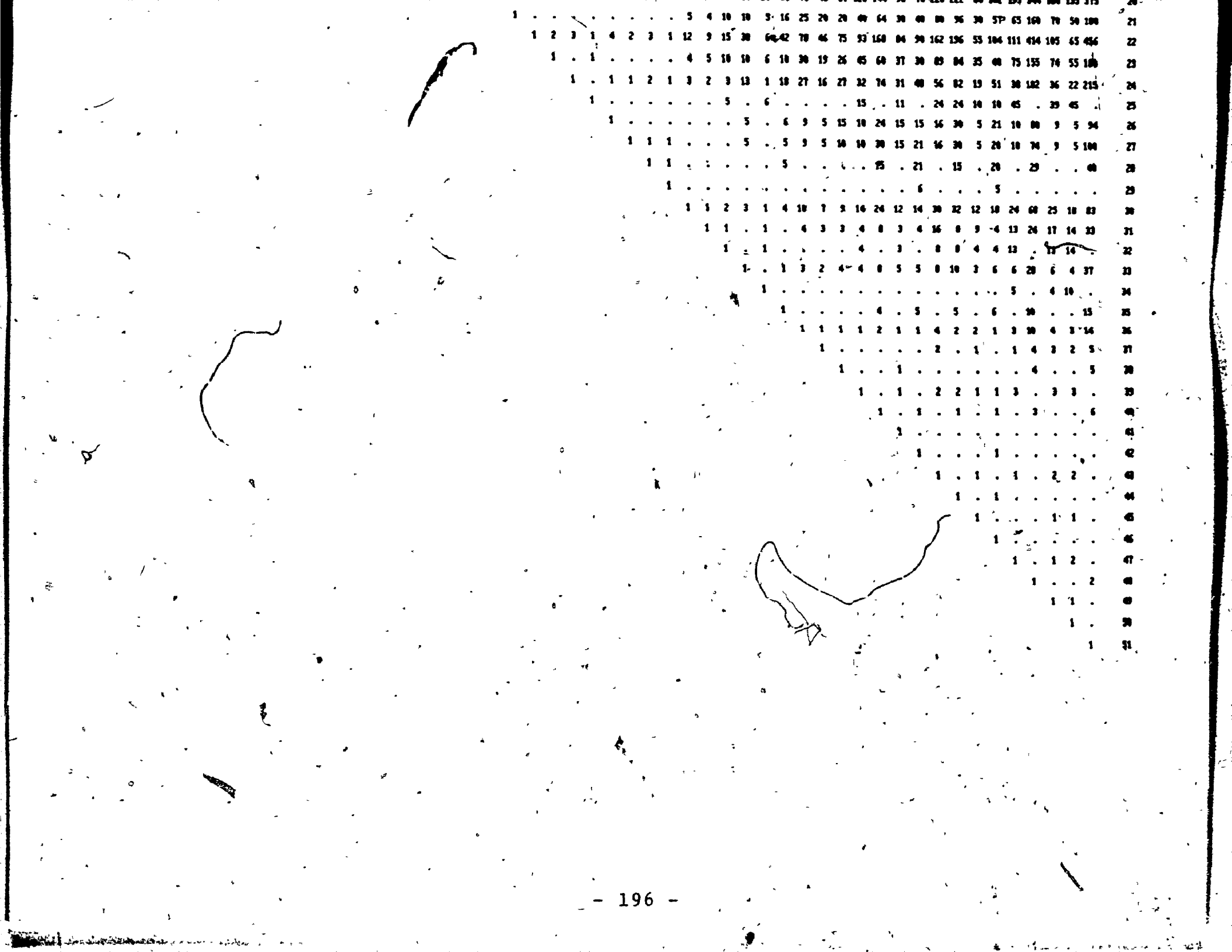


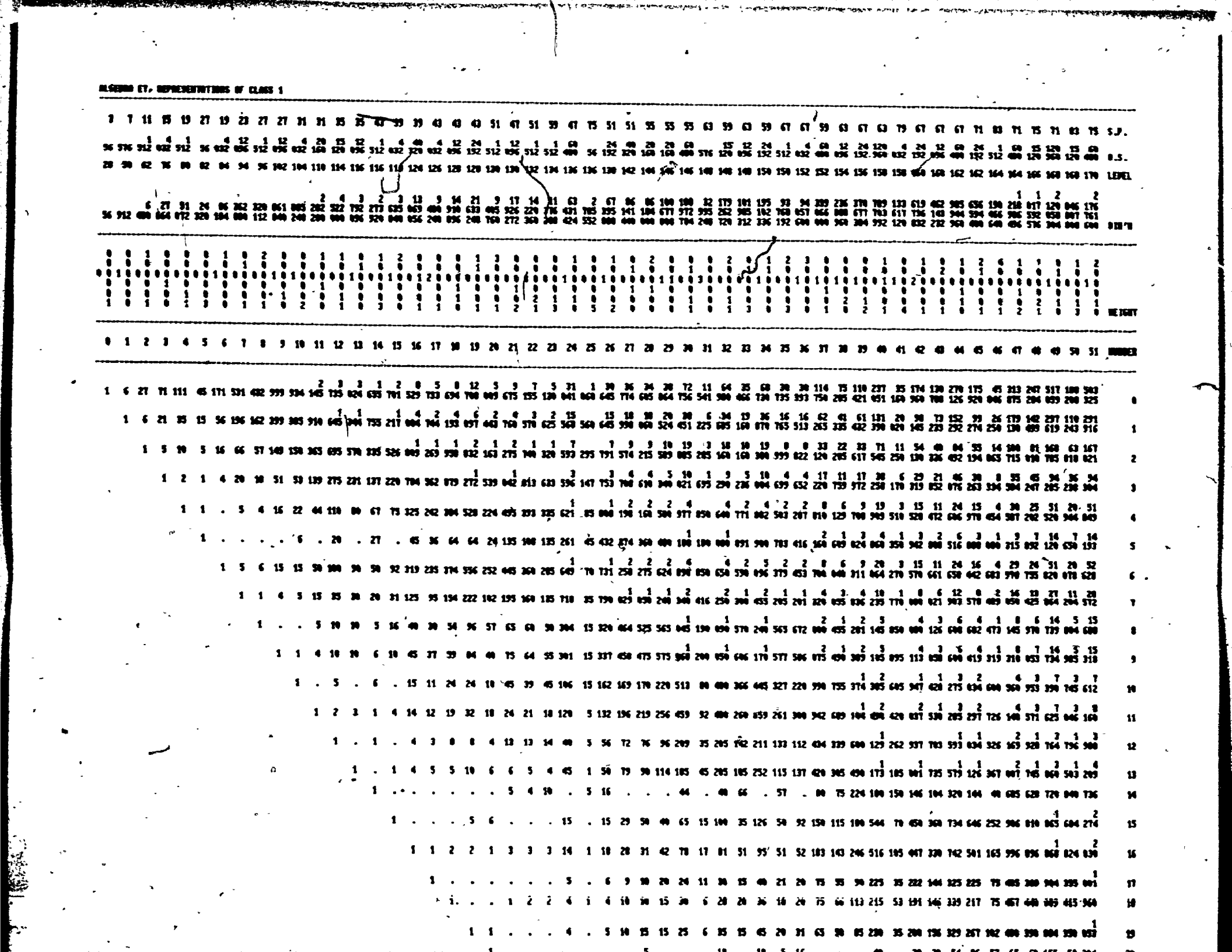


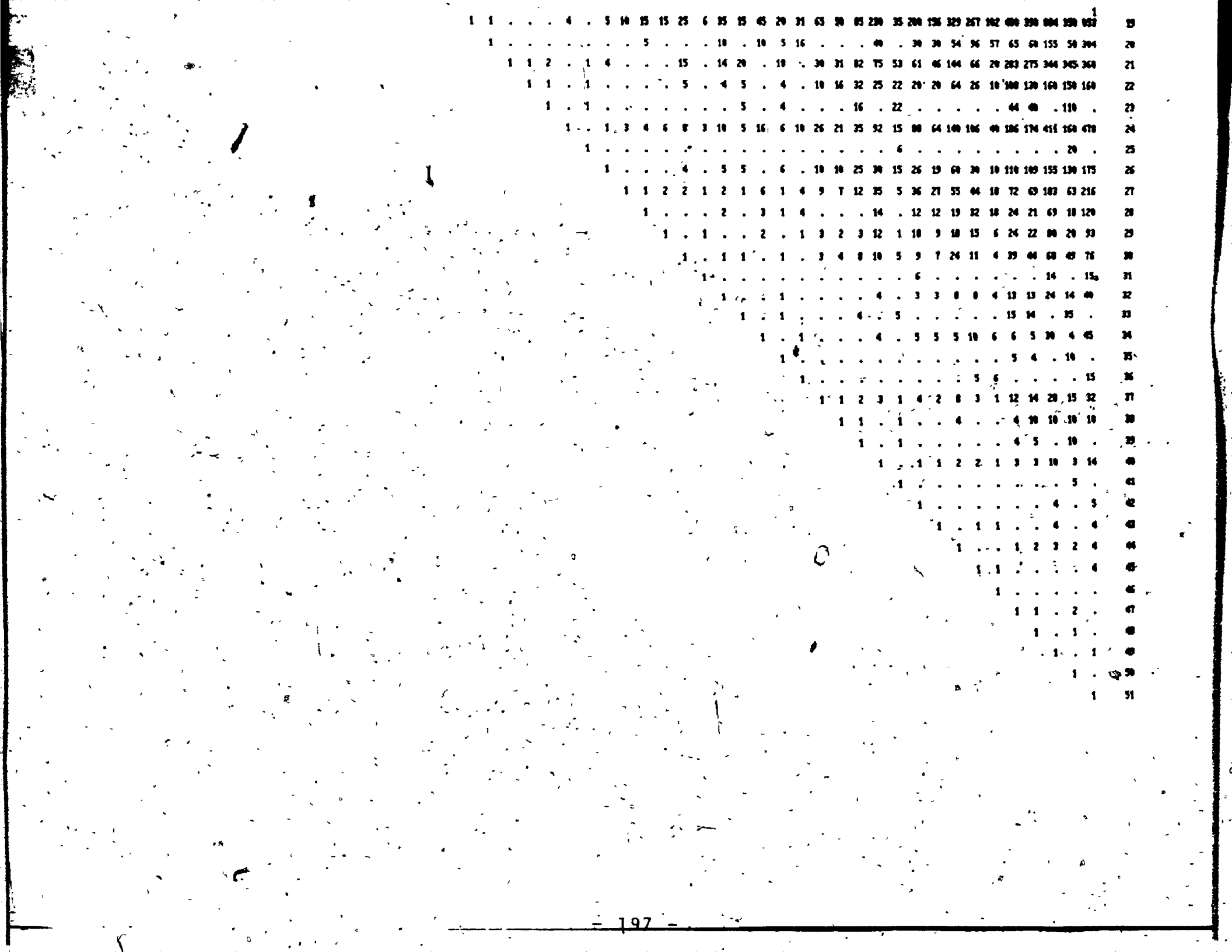

S.P.
B.S.
LEVEL
WEIGHT
NUMBER

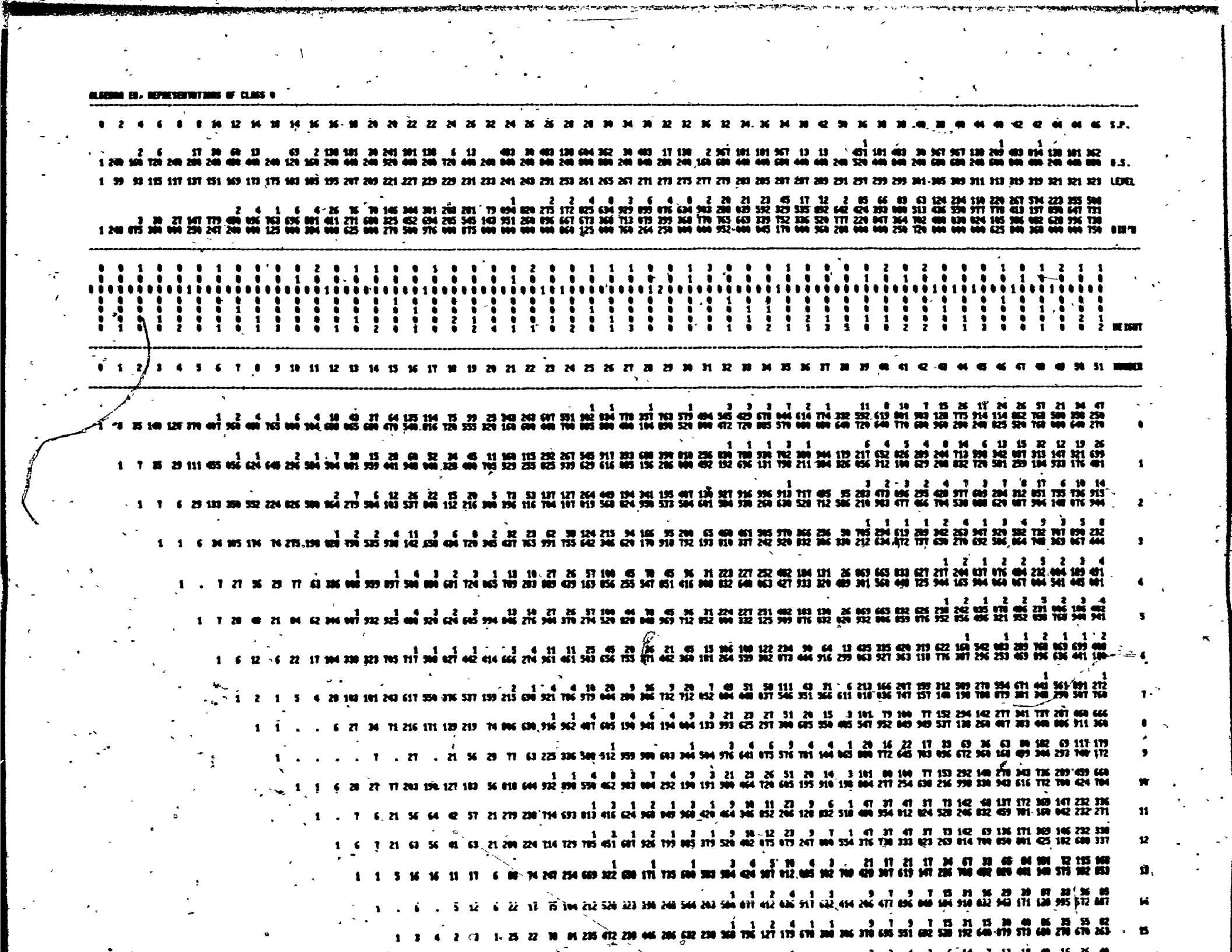
ALGEBRA ET, REPRESENTATIONS OF CLASS 0

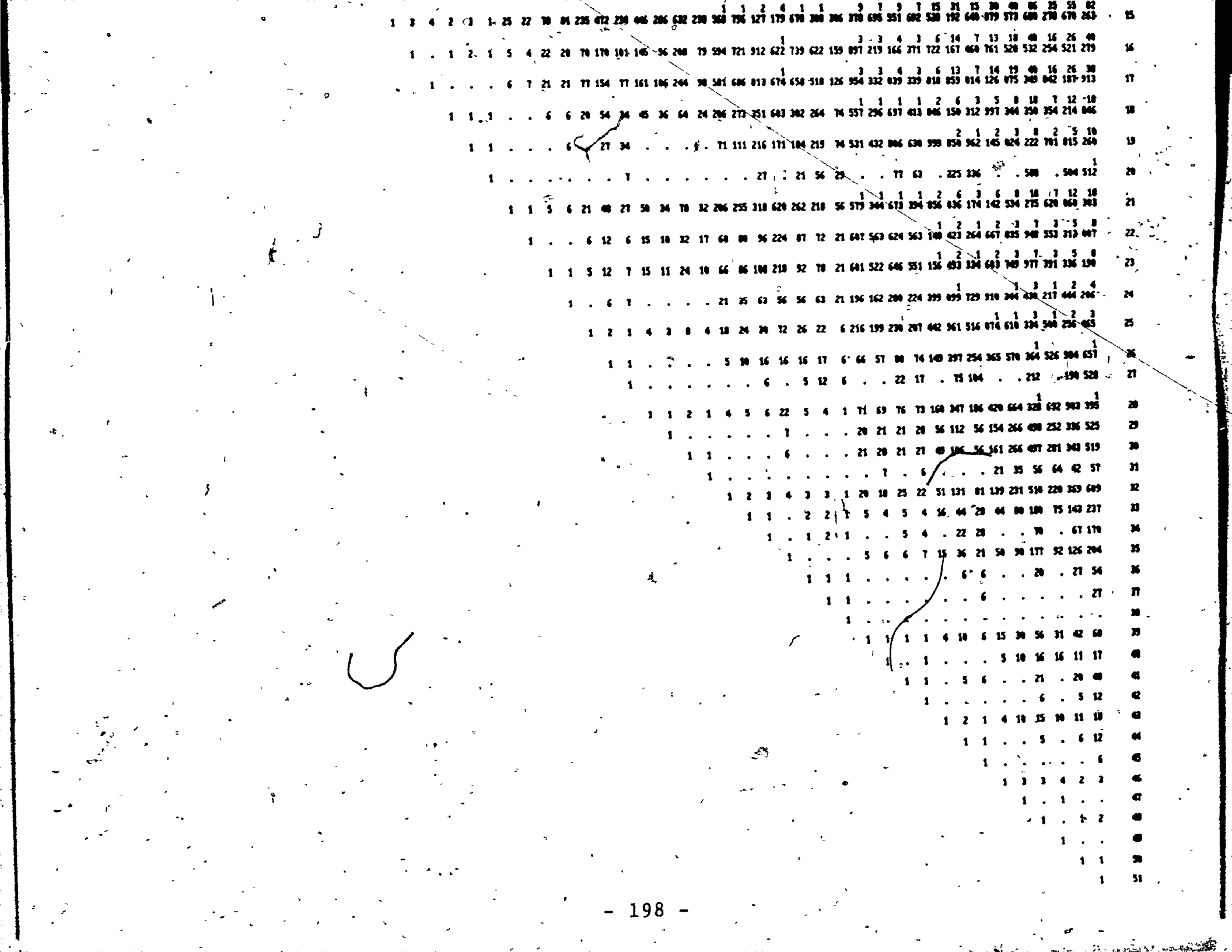





ALGEBRAIC REPRESENTATIONS OF CLASS 1
S.P.
B.S.
LEVEL
DIFF
WEIGHT
NUMBER



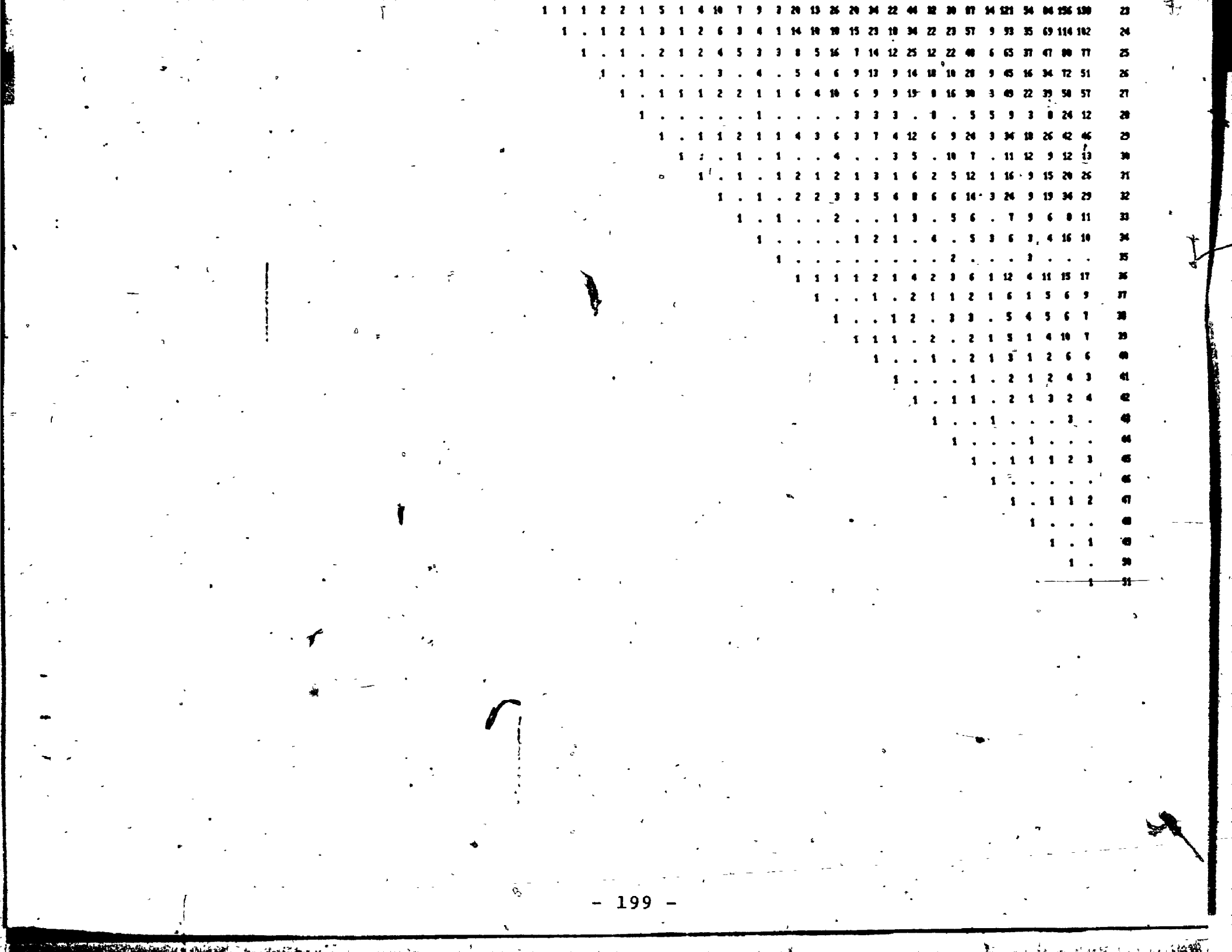

ALGEBRA EB, REPRESENTATIONS OF CLASS 0
S.P.
0.S.
LEVEL
DIM'N
WEIGHT
MULTIPLICITY



ALGEBRA F4, REPRESENTATIONS OF CLASS 0

S.P.	0	2	4	6	8	10	12	16	14	18	18	20	22	24	26	26	28	32	36	30	34	34	38	36	38	40
O.S.	1	24	24	96	24	144	96	24	192	24	288	144	288	96	144	192	192	24	24	576	144	288	288	288	192	144
LEVEL	1	17	23	26	30	39	43	45	47	49	53	55	59	61	61	63	65	65	67	69	71	73	75	75	77	77
DIM'N	1	26	52	273	324	1 053	1 274	1 053	4 056	2 632	8 424	10 829	19 278	19 448	17 901	24 749	29 172	16 282	12 376	106 456	76 076	107 406	119 119	160 056	205 751	160 056
WEIGHT	00>00	00>01	10>00	00>10	00>02	10>01	01>00	20>00	00>11	00>03	10>10	10>02	01>01	00>20	20>01	00>12	11>00	00>04	30>00	10>11	10>03	01>10	20>10	01>02	00>21	20>02
NUMBER	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	1	2	4	9	12	21	26	21	64	36	96	125	176	176	141	205	220	126	80	704	652	606	697	808	1 053	816
1		1	1	5	5	14	13	9	40	26	66	77	126	117	106	201	147	84	53	512	344	459	467	645	800	589
2			1	2	3	6	10	8	24	13	40	56	80	84	66	130	110	62	47	360	239	130	330	456	592	463
3				1	1	4	4	3	14	9	27	32	57	54	50	95	72	41	27	262	181	247	258	351	444	329
4					1	1	3	3	8	5	15	23	33	39	29	64	30	32	24	104	122	173	101	267	324	269
5						1	1	1	4	3	10	12	24	23	23	42	36	19	14	120	92	120	139	104	230	179
6							1	1	2	1	5	9	13	36	12	27	20	14	13	80	59	87	96	140	169	141
7								1	.	.	2	3	4	6	6	9	14	6	8	40	26	42	52	60	83	70
8									1	1	3	4	9	9	9	10	14	9	6	59	45	63	71	50	126	93
9										1	.	1	3	3	3	8	5	5	3	24	23	28	32	45	63	45
10											1	1	3	3	4	6	6	3	3	26	20	30	37	44	60	40
11												1	1	2	1	4	5	3	3	16	12	18	22	34	41	37
12													1	1	1	2	2	1	1	10	9	13	16	20	29	22
13														1	.	1	2	1	1	6	4	8	9	15	19	18
14															1	.	1	.	1	4	3	5	10	7	11	12
15																1	.	1	.	3	4	5	6	9	13	9
16																	1	.	1	2	1	3	5	6	7	9
17																		1	.	.	1	.	.	3	3	3
18																			1	.	.	.	2	.	.	3
19																				1	1	2	3	3	5	4
20																					1	.	.	1	2	1
21																						1	1	1	2	1
22																							1	.	.	1
23																								1	1	1
24																									1	.

S.P.	42	42	50	44	50	40	46	52	52	64	50	54	54	54	56	58	58	62	66	60	72	62	68	66	66	68
O.S.	192	576	24	288	144	96	576	192	144	24	576	96	576	288	192	144	576	192	288	576	24	576	144	288	576	288
LEVEL	79	81	81	83	83	85	85	87	87	89	89	91	91	91	93	93	95	95	97	97	97	99	99	101	101	103
DIM'N	212 992	279 040	81 081	420 147	184 756	226 746	307 644	340 119	412 776	180 776	1 110 200	629 356	1 327 104	952 952	1 341 522	1 082 456	1 001 371	1 042 899	1 076 944	2 752 556	342 056	3 321 372	1 404 406	2 400 563	4 313 088	3 500 536
WEIGHT	00>13	11>01	00>05	10>20	30>01	02>00	10>12	21>00	10>04	40>00	01>11	00>30	20>11	01>03	00>22	20>03	11>10	00>14	30>10	11>02	00>06	10>21	30>02	02>01	10>13	01>20
NUMBER	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
0	1 024	1 636	345	1 003	704	906	3 100	1 263	1 560	336	4 032	2 116	4 416	3 200	4 306	3 056	5 355	3 147	2 064	8 140	904	10 836	3 538	6 339	11 320	8 740
1	776	1 304	269	1 365	567	674	2 475	536	1 153	230	3 116	1 691	3 504	2 571	3 464	2 520	4 340	2 520	2 352	6 464	745	8 011	3 106	5 234	9 216	7 090
2	504	564	190	1 074	416	552	1 904	704	554	212	2 400	1 324	2 708	1 995	2 764	1 950	3 436	2 900	1 060	5 289	610	7 076	2 552	4 200	7 472	5 060
3	430	755	156	799	330	404	1 467	570	720	140	1 940	1 050	2 170	1 593	2 160	1 590	2 774	1 600	1 531	4 153	402	5 718	2 036	3 454	6 042	4 716
4	320	545	116	623	242	331	1 116	477	575	134	1 504	813	1 704	1 220	1 724	1 221	2 167	1 212	1 204	3 302	400	4 354	1 634	2 746	4 072	3 073
5	240	427	80	456	200	237	840	340	426	94	1 160	630	1 332	963	1 331	509	1 143	1 001	900	2 620	309	3 656	1 322	2 250	3 900	3 052
6	176	301	63	354	140	194	637	289	339	85	890	489	1 034	732	1 047	747	1 308	701	771	2 133	253	2 806	1 101	1 776	3 128	2 529
7	80	164	33	195	84	114	352	177	192	58	512	204	624	422	610	452	830	471	500	1 326	150	1 796	717	1 120	1 960	1 624
8	130	232	31	252	113	134	400	190	249	56	670	382	795	500	806	683	1 072	623	624	1 633	199	2 301	842	1 441	2 409	1 990
9	72	120	32	133	61	73	264	111	146	34	304	224	456	346	404	363	630	392	300	992	132	1 422	523	905	1 504	1 263
10	64	125	26	136	66	75	262	116	130	35	306	221	472	335	470	362	654	370	350	1 000	121	1 423	537	913	1 562	1 272
11	40	80	19	103	41	61	191	97	113	32	200	163	352	240	363	263	406	294	296	803	100	1 059	441	697	1 240	1 029
12	32	63	14	70	34	40	139	63	76	20	214	126	266	193	273	212	304	222	239	597	75	863	320	566	960	797
13	22	39	9	53	20	34	99	54	60	19	150	92	198	176	211	150	204	163	170	400	62	660	271	429	756	642
14	12	33	6	36	23	21	69	35	37	14	112	67	160	100	144	124	230	119	160	352	42	513	215	347	572	400
15	18	28	9	32	16	19	72	30	45	10	114	70	141	111	159	120	216	140	130	346	51	510	192	340	600	406
16	8	19	3	27	12	10	40	33	30	13	82	49	114	70	111	84	166	90	113	282	34	307	174	258	444	393
17	8	5	5	9	3	8	24	12	23	6	40	24	40	45	71	45	77	72	52	155	32	213	80	140	200	220
18	.	4	.	6	6	4	9	14	6	8	16	11	40	13	24	26	54	20	52	90	10	122	70	84	136	133
19	6	14	3	16	9	10	34	17	20	6	59	37	80	56	82	67	126	71	86	199	26	290	119	206	346	295
20	4	5	3	6	3	4	16	7	12	3	20	10	36	34	47	37	61	40	42	107	19	166	63	117	212	171
21	2	6	1	7	4	5	15	8	9	3	30	20	40	28	40	35	70	37	40	130	14	100	67	122	150	176
22	.	3	.	3	4	2	6	6	3	3	12	8	26	11	17	20	40	15	37	60	6	52	40	68	104	90
23	2	2	1	5	1	4	10	7	9	3	20	13	26	20	34	22	44	12	30	87	14	121	54	84	156	130
24	1	2	1	3	1	2	6	3	4	1	14	10	10	15	23	10	34	22	23	57	9	93	35	69	114	102



ALGEBRA G2. REPRESENTATIONS OF CLASS 0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
S.P.	0	2	6	8	14	18	24	26	32	38	42	50	54	56	62	72	74	78	86	96	98	98	104	114	122	128
O.S.	1	6	6	6	12	6	6	12	6	12	12	6	6	12	12	6	12	12	12	6	6	12	12	12	12	6
LEVEL	1	7	11	13	17	19	21	23	25	27	29	31	31	33	35	37	37	39	41	41	43	43	45	47	47	49
DIM'N	1	7	14	27	64	77	77	189	182	286	448	378	273	729	924	714	896	1547	1728	748	1254	2079	2926	3003	2261	2079
WEIGHT	0 < 0	1 < 0	0 < 1	2 < 0	1 < 1	3 < 0	0 < 2	2 < 1	4 < 0	1 < 2	3 < 1	5 < 0	0 < 3	2 < 2	4 < 1	6 < 0	1 < 3	3 < 2	5 < 1	0 < 4	7 < 0	2 < 3	4 < 2	6 < 1	1 < 4	8 < 0
NUMBER	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	1	1	2	3	4	5	5	9	8	10	16	12	9	21	24	18	20	35	36	16	24	39	52	51	35	33
1		1	1	2	4	4	3	8	7	10	14	11	7	19	23	16	20	32	34	13	23	37	50	48	35	31
2			1	1	2	3	3	6	5	7	12	9	7	16	19	14	16	29	30	13	20	33	44	44	30	20
3				1	2	2	2	5	5	7	10	8	5	15	18	13	16	26	28	11	19	31	43	41	30	27
4					1	1	1	3	3	5	7	6	4	11	14	10	13	21	23	9	16	26	36	35	26	23
5						1	1	2	2	3	6	5	4	9	11	9	10	19	20	9	14	23	31	32	22	21
6							1	1	1	2	4	3	3	7	8	7	8	15	16	8	11	19	26	27	19	18
7								1	1	2	3	3	2	6	8	6	8	13	15	6	11	18	25	25	19	17
8									1	1	2	2	1	4	6	5	6	10	12	5	9	14	21	21	16	15
9										1	1	1	1	3	4	3	5	8	9	4	7	12	17	17	14	12
10											1	1	1	2	3	3	3	7	8	4	6	10	14	16	11	11
11												1	.	1	2	2	2	4	6	2	5	7	11	12	9	9
12													1	1	1	1	2	4	4	3	3	7	9	10	8	7
13														1	1	1	2	3	4	2	3	6	9	9	8	7
14															1	1	1	2	3	1	3	4	7	8	6	6
15																1	.	1	2	1	2	2	4	6	3	5
16																	1	1	1	1	1	3	4	4	5	3
17																		1	1	1	1	2	3	4	3	3
18																			1	.	1	1	2	3	2	3
19																				1	.	1	1	1	2	1
20																					1	.	1	2	1	2
21																						1	1	1	2	1
22																							1	1	1	1
23																								1	.	1
24																									1	.
25																										1

	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
S.P.	126	134	150	146	152	162	158	168	182	182	186	200	194	216	206	218	222	224	242	234	254	248	230	266	266	294
O.S.	12	12	6	12	12	6	12	12	12	12	12	6	12	6	12	12	12	12	6	12	12	12	12	12	12	6
LEVEL	49	51	51	53	53	55	55	57	57	59	59	61	61	61	63	63	65	65	67	67	67	69	69	71	71	71
DIM'N	4096	5103	1729	4928	4914	3289	7293	8372	4928	7722	9177	5985	12096	3542	13090	10206	11648	15625	7371	19019	9660	19683	18304	17017	24948	6630
WEIGHT	3 < 3	5 < 2	0 < 5	7 < 1	2 < 4	9 < 0	4 < 3	6 < 2	1 < 5	8 < 1	3 < 4	10 < 0	5 < 3	0 < 6	7 < 2	2 < 5	9 < 1	4 < 4	11 < 0	6 < 3	1 < 6	8 < 2	3 < 5	10 < 1	5 < 4	0 < 7
NUMBER	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
0	64	75	25	68	66	43	93	104	50	[illegible]	105	55	132	38	136	102	116	151	63	179	94	177	180	145	270	54
1	60	72	22	66	63	41	91	99	50	87	100	53	128	34	133	99	112	140	67	173	84	172	154	142	265	50
2	56	66	22	68	58	30	[illegible]	93	50	81	95	43	120	34	124	93	106	130	63	165	77	163	140	134	195	50
3	52	63	19	58	56	36	[illegible]	89	50	70	90	48	116	31	121	90	102	136	61	159	77	159	142	131	190	46
4	45	55	11	51	40	32	72	79	45	70	81	43	105	28	110	82	93	124	56	146	71	146	131	121	176	43
5	42	38	17	46	45	30	63	74	40	65	77	40	98	28	102	77	88	115	53	139	63	130	125	114	167	43
6	36	43	15	40	40	26	57	66	36	58	69	36	88	26	92	79	80	105	48	127	60	127	116	105	154	40
7	33	41	13	39	38	25	56	62	36	56	63	35	85	23	90	68	77	103	47	122	60	123	111	103	150	37
8	28	35	11	34	33	22	48	55	32	50	58	32	76	21	81	61	70	94	43	111	55	113	102	95	138	34
9	24	30	10	29	29	19	43	40	29	44	52	28	68	19	73	56	63	65	39	101	51	103	94	87	127	32
10	22	27	10	26	26	18	38	45	25	41	48	26	63	19	67	52	60	70	37	96	46	97	90	82	120	32
11	16	22	7	22	21	15	32	37	22	35	40	23	54	15	59	45	52	63	33	83	42	86	10	74	107	27
12	16	19	8	18	20	13	29	34	20	31	39	20	50	16	53	43	48	64	30	79	39	88	76	68	101	28
13	14	18	6	18	18	12	28	32	20	30	36	20	48	14	52	41	46	63	29	75	39	78	72	67	58	25
14	11	15	5	16	15	11	24	28	17	27	31	19	42	12	47	36	42	56	27	68	35	71	65	62	60	23
15	8	11	4	12	11	9	17	23	12	22	25	15	34	11	38	29	36	45	21	58	28	61	56	53	76	21
16	8	10	4	10	12	7	10	20	14	19	25	13	33	10	36	30	32	46	21	55	30	57	55	50	75	20
17	7	9	4	9	10	7	15	19	11	18	23	12	30	10	33	27	31	41	20	52	26	54	52	47	70	20
18	4	7	2	8	7	6	12	15	9	16	17	11	25	7	29	22	27	35	18	44	23	48	48	43	61	16
19	4	4	3	4	7	3	9	11	8	10	16	7	20	8	21	20	20	30	13	37	20	38	40	33	53	16
20	2	4	1	6	4	5	8	11	6	12	12	9	10	5	23	16	22	27	15	35	18	39	34	36	40	13
21	3	4	2	4	6	3	9	10	8	10	14	7	19	6	21	19	19	29	13	35	20	37	37	33	51	14
22	2	3	1	4	4	3	7	9	6	9	11	7	16	5	19	15	18	25	12	31	17	34	32	31	45	12
23	1	2	1	3	2	3	4	7	3	8	8	6	12	4	15	11	16	18	11	26	12	29	26	27	37	11
24	1	1	1	1	3	1	4	4	5	4	8	3	10	4	11	12	10	18	7	21	14	22	25	20	34	10
25	.	1	.	2	1	2	2	4	2	6	4	5	8	2	11	7	12	13	9	18	9	23	18	22	20	7
26	1	1	1	1	2	1	3	4	3	4	7	3	9	4	10	10	10	15	7	20	11	21	23	19	31	10
27		1	.	1	1	1	2	3	2	4	4	3	7	2	9	7	9	12	7	16	9	19	17	18	26	7
28			1	.	1	.	1	1	2	1	4	1	4	3	4	7	4	9	3	11	8	11	16	10	20	8
29				1	.	1	1	2	1	3	2	3	4	1	7	4	8	8	6	12	6	15	12	16	19	5
30					1	.	1	1	2	1	3	1	4	2	4	6	4	9	3	10	8	11	14	10	19	6
31						1	.	1	.	2	1	2	2	1	4	2	6	4	5	8	3	11	8	12	13	4
32							1	1	1	1	2	1	3	1	4	4	4	7	3	9	6	10	11	10	16	5